IMPROVING MEMORY AND I/O SYSTEMS THROUGH FORESIGHT

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Janani Mukundan

January 2014

IMPROVING MEMORY AND I/O SYSTEMS THROUGH FORESIGHT

Janani Mukundan, Ph.D.

Cornell University 2014

Traditionally, DRAM scheduling techniques have been optimized for performance. Only recently has there been a push for improving other optimization metrics, such as energy efficiency, power, or fairness. A multitude of scheduling algorithms have been proposed in the past few years for tackling these goals. But a major shortcoming in many of these techniques is that they are made up of inflexible, static hard-coded scheduling policies that lack the ability to learn and improve automatically with experience, or to reconfigure themselves to target a variety of such optimization metrics.

Recently, İpek et al. [32] proposed the use of reinforcement learning (RL) to design high-performance, self-optimizing memory schedulers. Reinforcement learning is a machine learning technique that learns automatically with experience, by interacting with the environment. It tries to pick the actions that maximize a desired long-term objective function. By using an online learning technique like RL, memory controllers have the capability of foresight and long-term planning, thereby enabling a non-greedy approach to scheduling. However, İpek et al.'s methodology has a key limitation: it does not possess a generalizable way to target an objective function.

In my thesis, we present a framework for designing a class of memory controllers that have the capability of managing multiple objective functions in a synergistic and coordinated fashion. MORSE (MultiObjective Reconfigurable Self-Optimizing Scheduler) is a systematic and general methodology to design

reconfigurable DRAM schedulers following RL principles. Our framework also provides a way to reconfigure the scheduler on the field (post-silicon), whether at boot time or dynamically at run time, to accommodate changes to the optimization criteria.

Beyond DRAM scheduling, we find that the storage technology landscape is rapidly undergoing many changes, primarily enabled by device scaling. In particular, DRAM is scaling in terms of density and frequency. High-density DRAM chips are becoming increasingly more common. As a result, memory systems are becoming more complex structurally. Due to this, a number of problems that were either non-existent or inconsequential in prior DRAM systems, have started surfacing.

In particular, DRAM refresh overheads are on the rise. In the next part of my thesis, we investigate refresh overheads that are caused due to DRAM scaling. We propose simple scheduling techniques that help mitigate refresh stalls that occur in high density DDR4 memory systems. These techniques again involve the notion of foresight, by anticipating the patterns that lead to refresh stalls, and planning ahead of time to mitigate them. Scheduling refreshes is a real-time algorithm, and missing deadlines may lead to reliability concerns. Hence, this research initially focuses on simple prioritization techniques that do not require complex online learning to overcome refresh stalls.

Over the past few years computer systems of all types have started integrating flash memory. The usage of NAND-flash based SSDs is becoming more widespread. As NAND based flash scales, flash memory's high density and low cost make it a viable option for desktop and high-end server environments. Just like DRAMs, there are a number of interrelated goals and metrics that need to be managed synergistically in the SSD domain as well. Therefore, in the fi-

nal chapter of my thesis, we tackle the problem of improving scheduling in I/O systems by leveraging our RL based framework for designing self-optimizing schedulers. Current I/O controllers manage goals like write placement, garbage collection, and wear leveling individually. These schedulers also don't have the capability of online learning: they are fixed, static scheduling policies. Since NAND-flash characteristics are known to vary over time as the flash dies start wearing out, it is important to understand how these techniques correlate with each other. We adopt the principles of reinforcement learning to build self-optimizing SSD controllers that have the capability of foresight and planning, and can synergistically manage multiple objective functions in I/O systems.

BIOGRAPHICAL SKETCH

Janani Mukundan attended the University of Madras as a Computer Science undergraduate student from the year 1999 to 2003. Right after her graduation from the University of Madras, she attended North Carolina State University as an MS student in Computer Science. After completing her masters at NC State, she worked as a component design engineer at Intel. She quit Intel in 2007 to join the graduate program at Cornell University, pursuing a PhD in Computer Engineering. Through her years at Cornell, she worked with her advisor, Prof. José Martínez, on various topics in the computer architecture field, including designing reconfigurable self-optimizing architectures for memory and I/O systems and improving the performance of the processor, memory and storage subsystems using machine learning techniques.

This dissertation is dedicated to my parents, my brother, and my dear friends.

ACKNOWLEDGEMENTS

Completing my PhD degree has probably been the most challenging and fulfilling activity of the first three decades of my life. The best and worst moments of my doctoral journey have been shared with many people. It has been an honor and great privilege to spend several years in the Department of Electrical and Computer Engineering at Cornell University. I will always cherish my memories at CSL and this experience will always remain close to my heart.

This journey would not have been possible without the love and support of my mom, dad and my brother. I want to thank my family for their affection and encouragement. I love them very much and I am deeply indebted to them.

My deepest and heartfelt gratitude and appreciation goes to my advisor, Prof. José Martínez. He patiently provided the vision, encouragement and advise necessary for me to proceed through the doctoral program and complete my dissertation. He has taught me, both consciously and unconsciously, how good research is done. He has supported me during my darkest and my brightest days at Cornell. José has helped me become a better researcher and a better human being. I thank him for that. I appreciate all his contributions of time, ideas, and funding to make my Ph.D. experience productive and stimulating.

Next, I would like thank my thesis committee for providing me insightful comments and suggestions to improve my dissertation work. Their guidance and assistance were an enormous help to me.

I want to thank my dearest friends (in no particular order), Swapnaa, Shyam, Indhu, Rama and Sri. They have been there for me when I most needed them. Their impact in my life has been huge, and I am a better individual because of them. Words cannot express my gratitude, love and affection for them.

Special thanks to my friends and colleagues, Max, Saugata, the twins – Meyrem and Nevin, Sarvani, Sowmya, Prajna and Xiaodong. Discussions with them, both professional and personal, have been very insightful and illuminating. I have benefited greatly due to their presence. Without their support, encouragement, guidance and tolerance this dissertation would not have materialized. A word of appreciataion to Swami for being kind, patient and helpful. Special thanks also to Naomi Salama, Crookie bear and Adzuki, for being a delight in my life.

I would like to express my appreciation to my colleagues and co-authors, Engin, Hillery, KH, Michele and Ashish. I am grateful for the feedback and constructive comments they have offered.

I want to end by thanking my family again. For my parents, who raised me to believe and fight for liberty and freedom of thought and expression, and who have supported me wholeheartedly in all my pursuits. For my brother, who has been patient, caring and affectionate during these years. Thank you.

Janani Mukundan

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 SECTION 1

The storage technology landscape is rapidly undergoing many changes, primarily enabled by device scaling. As a result, memory and I/O systems are becoming complex structurally. This introduces a number of problems that were either easily overlooked or did not surface in prior technology generations. In my thesis, I analyze three of these problems as shown below. Specifically, I look at DRAM and NAND based flash storage technologies because of their widespread and prevalent usage in current day memory systems: there is high volume production of memory chips based on both DRAM and flash in the industry. Moreover, they are likely to yield insights that are portable to other speculative memory technologies like PCM, MRAM etc., as well.

1.1.1 Memory scheduling for diverse optimization functions

Traditionally, computer architects have primarily optimized DRAM controllers for performance. This was appropriate, as the gap between the CPU and memory speed kept growing at the time. DRAM energy consumption has been given due consideration only relatively recently. In current and upcoming multicore-based servers, DRAM accounts for a significant fraction of power consumption. Therefore, apart from performance, power and energy are also becoming first order issues while designing memory schedulers for multicore systems. In addition to these issues, DRAM memory bandwidth is a critical shared resource in

a multi-core system, and it is important to efficiently share the memory bandwidth among multiple threads running in a multicore environment, so as to not adversely affect system throughput and fairness. As DRAM scaling continues, modern memory systems have reached an inflection point. Partly due to this scaling, there is a convergence of several trends in modern memory systems, such as the increasing importance of energy consumption, the need for QoS and Fairness etc., that need to be addressed immediately.

Contributions

– We propose MORSE, a systematic and general mechanism to designing self-optimizing DRAM schedulers that can target arbitrary figures of merit and has the capability of foresight and long-term planning.

– We employ genetic algorithms to automatically calibrate the relative importance that the scheduler places on the different DRAM actions for a given environment and objective function.

–We also employ a multi-factor variation of feature selection that takes into account first-order interactions among system attributes, which are used by the scheduler to sense the systems state at each point in time.

Importantly, the resulting hardware need not directly observe the objective function on the field: only during training at design time (using simulation models) does our framework require the objective function to be observable. This allows our framework to target relatively sophisticated figures of merit that would be generally hard to measure on the field (e.g., weighted speedup).

1.1.2 Refresh in High Density DRAMs

Since DRAMs are capacitive in nature, the dynamic nature of DRAM requires logic to carefully track the DRAM lines that need to be refreshed, and to issue refresh commands in a timely fashion. For a long time, refresh commands were relatively short and infrequent, and thus system performance and power were not significantly impacted. As DRAM density increased, industry decided to design for a constant per-cell retention time (64 ms) and refresh interval (t_{REFI}), changing instead the time that each refresh command takes to complete (t_{RFC}). This essentially means that each refresh command refreshes a larger number of rows as chip density increases. As refresh latency (t_{RFC}) increases, concern has been raised about the performance impact of refresh.

Additionally, as technology scaling advances, DRAM tends to maintain the same number of electrons in its storage capacitor, so DRAMs can theoretically continue to scale. However, concerns about the end of scaling remain well-founded, since it will be more and more difficult to maintain the same amount of charge in the storage capacitor in future technologies. There are several reasons for this, and we list a few here. (a) decrease in per-cell capacitance for smaller DRAM cells, (b) supply voltage scaling to meed low power demands causes increased leakage, (c) reduced sensitivity of the sense amps etc. All of these scaling issues make it more difficult to meet the JEDEC DRAM cell retention time specification of 64 ms in future technology. Thus, in addition to increases in t_{RFC} as DRAM chip density increases, it is anticipated that t_{REFI} may also worsen.

Contributions

– We conduct an analysis of the recently introduced DDR4 DRAM's Fine Granularity Refresh (FGR), and determine that there is no single mode that works well across all the applications studied.

– We propose *Adaptive Refresh*, a simple yet highly effective mechanism that leverages DDR4 DRAM's FGR, by dynamically choosing the mode that suits best each application, and each phase within the application.

– We introduce *Command Queue Seizure*, a phenomenon that we show will become a concern in systems built with high-density DRAM chips (16 and 32 Gb), which are expected to be common for DDR4 DRAM and beyond.

– We propose *Delayed Command Expansion* (DCE) and *Preemptive Command Drain* (PCD), two complementary mechanisms to address the Command Queue Seizure phenomenon by proactively increasing the percentage of commands to non-refreshing ranks. This provides the scheduler more opportunities to issue commands, thereby reducing idle cycles and improving performance. Both DCE and PCD provide simple, yet effective forms of foresight and planning as they try to anticipate and prevent a command queue seizure ahead of time.

1.1.3 Improving I/O scheduling for FLASH-based solid state drives (SSDs) through Foresight

Over the past few years computer systems of all types have started integrating flash memory. Initially, because of its small size, low power consumption, and low-cost I/Os per second, flash was a natural fit for embedded devices. As NAND based flash scales, flash memory's high density and low cost make it a viable option for desktop and high-end server environments. However, as flash scaling continues, endurance of flash chips is projected to drop as well. Current systems do not have the capability to manage concurrency and endurance synergistically: The flash controllers in these systems manage performance, wear, and garbage collection individually. However, it is important to understand how these metrics relate to one another and how they impact each other as well. Therefore, designing flash controllers for modern I/O systems will require more sophisticated scheduling algorithms that have the capability of synergistic learning. To conclude, we investigate the effectiveness of using reinforcement learning (RL) in designing self-optimizing SSD controllers in hardware that can improve performance by using foresight to better leverage parallelism.

Contributions

– Investigate the technique of binning to perform write placement, garbage collection and wear leveling efficiently in hardware.

– Investigate the effectiveness of reinforcement learning in designing self-optimizing flash controllers, particularly for the following goals: performance, wear leveling and garbage collection.

CHAPTER 2

MORSE: IMPROVING SCHEDULING IN MEMORY SYSTEMS

2.1 Introduction

Modern high-performance memory subsystems support a high degree of concurrency. This is primarily accomplished by increasing the number of independent channels and/or increasing the number of independent banks in a channel [17, 18, 19, 31, 76]. It is critical that the memory controller be able to produce a schedule that can leverage this potential concurrency, all while abiding by numerous strict timing constraints imposed by the DRAM.

Traditionally, computer architects have primarily optimized DRAM controllers for performance [18, 29, 54, 53, 59, 76]. This was appropriate, as the gap between the CPU and memory speed kept growing at the time.

DRAM energy consumption has been given due consideration only relatively recently [20, 30]. In current and upcoming multicore-based servers, DRAM accounts for a significant fraction of power consumption [43]. Therefore, apart from performance, power and energy are also becoming first-order issues while designing memory schedulers for multi-core systems.

In addition to these issues, DRAM memory bandwidth is a critical shared resource in a multi-core system, and it is important to efficiently share the memory bandwidth among multiple threads running in a multicore environment, so as to not adversely affect system throughput and fairness.

Several scheduling algorithms have been proposed in the past to tackle the

problems listed above. Most such proposals are relatively inflexible in two ways: (1) they have a limited ability to adapt to the environment and to improve automatically with experience; and (2) they each target a particular objective function.

İpek et al. [32] propose the use of reinforcement learning (RL) [67] to design high-performance *self-optimizing* memory schedulers. Reinforcement learning works by interacting with the environment and learning automatically with experience to pick the actions that maximize a desired long-term objective function. İpek et al. show that, when used to target performance, this approach can outperform existing ad hoc designs by a significant margin.

Still, İpek et al.'s methodology has a key limitation: they do not propose a generalizable way to target an objective function (performance in their case). Because it is intuitive that bus utilization and throughput (and ultimately performance) correlate strongly for memory-intensive applications, it was natural and entirely appropriate for them to take a completely ad hoc approach to designing the RL reward function, by trivially rewarding load/store commands over precharge and activate commands. Unfortunately, this approach becomes much more difficult in other important scenarios that target more sophisticated objective functions (e.g., metrics that combine performance, energy, and/or fairness).

This work builds upon İpek et al.'s RL-based framework. **We propose MORSE, a *systematic* and *general* mechanism to designing self-optimizing DRAM schedulers that can target *arbitrary* figures of merit**. We employ genetic algorithms to automatically calibrate the relative importance that the scheduler places on the different DRAM actions for a given environment and

objective function (Section 4.4.1). We also employ a *multi-factor* variation of feature selection that takes into account first-order interactions among system attributes, which are used by the scheduler to sense the system's state at each point in time (Section 4.4.1). Importantly, the resulting hardware need not directly observe the objective function on the field: only during training at design time (using simulation models) does our framework require the objective function to be observable. This allows our framework to target relatively sophisticated figures of merit that would be generally hard to measure on the field (e.g., weighted speedup). Still, the hardware can be made to allow for on-the-field reconfiguration (Section 2.4).

Using this general approach, we rebuild İpek et al., and present quantitative evidence of significantly higher performance with respect to their original design (Section 2.6). We also use our general mechanism to design DRAM schedulers that can target energy efficiency (Section 2.7), as well as throughput/fairness of multiprogrammed workloads (Section 2.8). The designs prove significantly superior to the state of the art in each case.

2.2 Background

2.2.1 Power-Aware DRAM Interfaces

A basic DRAM interface has one or more DRAM channels; each channel consists of one or more memory modules. Most modern DRAM systems make use of dual in-line memory modules (DIMM); each DIMM consists of one or more

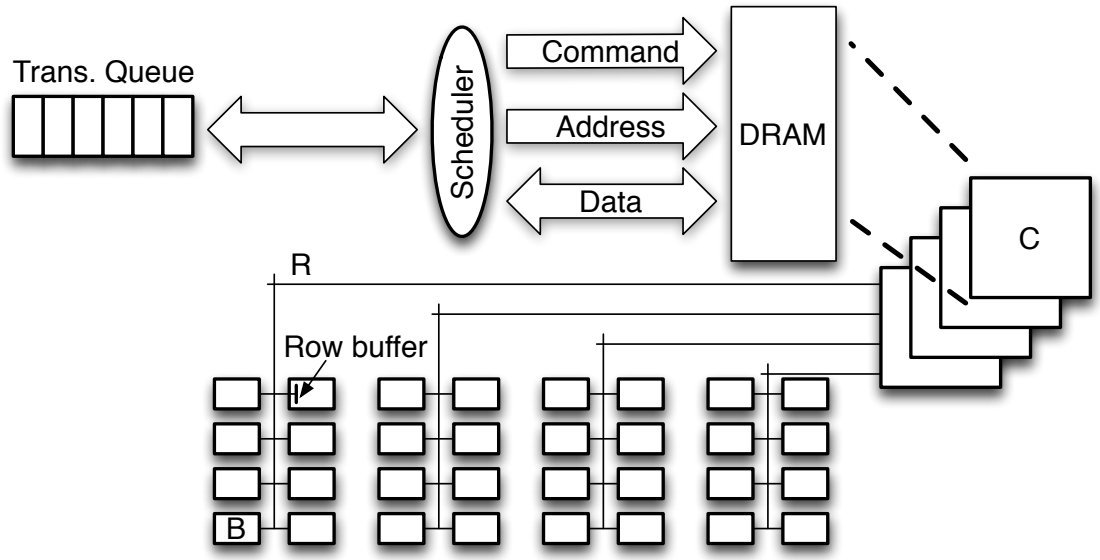

Figure 2.1: Basic DRAM Interface, with four independent channels (C), one quad ranked DIMM per channel (R), and eight internal banks (B) per rank.

ranks–a set of DRAM devices that operate in lockstep. Each DRAM device contains a set of independent memory arrays called *banks*. A bank is made up of *rows* (a.k.a. *DRAM pages*), which are simply a set of storage cells that are acted upon in parallel. Figure 2.1 shows a DRAM system interface, with four independent DRAM channels (C). Each channel has one quad ranked DIMM (R), with eight internal banks (B) per rank.

Read and *Write* commands to a bank can only take place to locations within one row at a time, which must be first copied into the bank's *row buffer* (*Activate* command). Prior to accessing a different row, the one currently stored in the row buffer must be written back to its permanent location (*Precharge* command). Finally, since DRAM is non-persistent, rows need to be periodically read out and restored to maintain data integrity (*Refresh* command). To make matters more difficult, modern DRAM chips have a large number of timing constraints that

must be obeyed when scheduling commands, which any memory scheduler must work around.

Current DDR3 SDRAMs have the capability of placing a rank in low-power mode. It may take as little as one DRAM cycle to place a DDR3 rank in low-power mode [5], however additional timing constraints must typically be met, depending on the DRAM command that is in progress. A rank in low-power mode must be powered back up before it can accept commands. Powering up a rank in DDR3 systems can take anywhere between 4-13 DRAM cycles [5]. Section 3.7.1 provides more details on the DDR3 DRAM interface that we model.

2.2.2 Basics of Reinforcement Learning

A reinforcement learning (RL) agent interacts with a probabilistic environment for the purpose of maximizing some notion of a long-term reward [67]. At each point in time, the agent does not necessarily pursue the action that offers the highest *immediate* reward; instead, the agent strives to take the action that provides the best *cumulative* reward over time. To learn how to do this, the agent needs to explore its environment carefully: Early exploitation (i.e., picking the action that seems most profitable in the long term at each point in time based on acquired knowledge) may result in an agent stuck with low-performing policies, while too much exploration (i.e., trying different actions) may cause the agent to take a long time to settle on an optimal policy. Moreover, the agent must never stop exploring completely if it is to adapt its policy to changes in the environment (e.g., program phases).

A basic RL model consists of: (1) a set of states that sufficiently describes the environment and the problem being solved; (2) a set of actions that the RL agent can perform; and (3) a reward function that assigns credit for performing an action in a state and moving to another state. In the context of DRAM scheduling, the RL agent is the memory scheduler, the pending requests and the state of the CPU and memory subsystem constitute the environment, and the legal DRAM commands at each point in time are the actions that the RL agent can perform [32]. The set of states and the reward function need to be determined depending on the long-term goal that needs to be achieved. At every time step: (1) the memory scheduler observes the state of the environment; (2) among the actions available for all the pending requests,[1] the memory scheduler chooses the one action that will maximize the cumulative reward; and (3) the memory controller performs that action, which results in a state change.

The agent needs to learn how to assign credit and blame for the actions it takes. A common way of learning to assign credit is through a technique called Q-learning. Formally, the Q-value of a state-action pair (s, a) while executing a policy π, $Q_\pi(s, a)$, is the expected cumulative reward resulting from taking action a in state s and following policy π thereafter. A Q-learning-based RL agent learns the optimal policy π^* indirectly, by learning $Q_{\pi^*}(s, a)$ for every state-action pair (s, a) (the *Q-value matrix*).

States are often represented as tuples of *attributes*. Because the size of the state space (in the case of Q-learning, the size of the Q-value matrix) is exponential in the number of attributes considered (this is often referred to as the

[1]Not all pending requests will have actions available at any point in time: For example, if a row has not yet been activated, a read to that row is not an available action. Even among available actions, only a subset may be evaluated in order to reach a decision every DRAM cycle.

"curse of dimensionality"), it is essential that the number of attributes and the resolution of each attribute be contained. This helps not only in reducing storage and speed requirements in a silicon implementation of the Q-value matrix; it also allows the RL agent to *generalize*, i.e., exploit knowledge acquired through past experience–in the case of Q-learning, approximate the Q-value of a previously unseen state-action pair (s, a) with the Q-value of state-action pair (s', a), with s and s' sufficiently close in the state space.

2.3 A General Framework For Self-Optimizing Memory Schedulers

In this section we describe how to generalize İpek et al.'s original RL-based memory scheduler design [32] to obtain high-quality schedulers that can target arbitrary objective functions–not just performance. We first present our design approach, and then describe a practical implementation.

2.3.1 Design

We now determine the three main characteristics of our RL-based design: actions, state attributes, and reward structure.

Available Actions

Concurrently to sensing the environment's state (Section 4.4.1), the scheduler determines whether a valid DRAM command exists for each pending memory request among:

Activate: Bring the contents of a bank's DRAM row into the bank's row buffer.

Precharge: Write the contents of a bank's row buffer back to the corresponding DRAM row. We categorize as a separate action the case where there are no active requests for a particular (open) row, yet the scheduler still may choose to preemptively precharge; we call this Preemptive Precharge.

Read(Load), Read(Store): Perform a read from a bank's row buffer.

Write: Perform a write to a bank's row buffer.

Rank Power Down (PwDn): Place the corresponding rank into a low power mode. When a rank is in low power mode, it cannot be accessed. Current DRAM subsystems already provide support for such low-power rank modes; in our implementation, we use those of the DDR3 interface [5].

Rank Power Up (PwUp): Bring the corresponding rank back to normal operation mode.

NoOp: If no legal DRAM command exists for this cycle (often due to DRAM timing constraints), the scheduler will do nothing and wait for the next cycle. (But a rank may remain powered down even if PwUp is a legal command.)

Reward Structure

In order to explore the environment, the scheduler implements an exploration mechanism known as ϵ-greedy action selection: Every DRAM cycle, with a small probability ϵ, the scheduler picks a random (legal) action; at all other times, it picks the (legal) action with the highest Q-value. This guarantees that there is a non-zero probability of visiting every entry in the Q-value matrix.

Each action is associated with an *immediate reward*. Once action a_t is picked and the immediate reward is determined, the Q-value prediction associated with the state-action pair (s_{t-1}, a_{t-1}) that was picked in the previous cycle $t-1$ can be updated using SARSA [67] as follows:

$$Q(s_{t-1}, a_{t-1}) \leftarrow (1-\alpha)Q(s_{t-1}, a_{t-1}) + \alpha[r_t + \gamma Q(s_t, a_t)]$$

where α is the *learning rate*, empirically determined;[2] r_t is the immediate reward collected for the action taken; and $0 \leq \gamma < 1$ is a *discount factor* that causes future rewards to be incorporated in the form of a geometric series.[3]

In the performance-oriented design of İpek et al. [32], the immediate reward function is picked solely based on expert intuition. Since the memory throughput (and ultimately execution time) of a memory-bound application tends to correlate strongly with the effective data bus utilization, the authors trivially assign an immediate reward of 1 to a read or write DRAM command, and an immediate reward of 0 to any other DRAM command. Unfortunately, this ap-

[2] A high learning rate quickly substitutes past knowledge with new information, whereas a small learning rate incorporates new knowledge slowly.

[3] Intuitively, γ can be interpreted as a knob that controls how important future rewards are relative to immediate rewards; larger γ values introduce more foresight at the expense of longer training times.

proach does not easily generalize: In a design that seeks to optimize a more sophisticated function (e.g., Et^2 or weighted speedup), an appropriate immediate reward function is not at all evident.

Automatic Derivation of Reward Structures

In this paper we propose to follow an automated approach to solve this problem. Specifically, we devise a genetic algorithm (GA) [51] to explore the search space of possible reward functions for a given objective function. [4]

Genetic algorithms (GAs) [51] are a heuristic search technique that is based on evolutionary processes. GA starts by randomly generating a population space of individuals, where each individual is a candidate solution for the problem being solved. Typically individuals in the population set are represented in binary as strings of 0's and 1's, but other encodings are also possible. Evolution is performed in generations and it starts by evaluating the fitness of the initial population according to an objective function. (Evaluation means simulations of candidate DRAM schedulers in our case.) Based on the fitness of individuals in the population set, the next generation of individuals are determined stochastically using some form of fitness based selection technique. These new individuals are then further evolved using operations like crossover and mutation, which leaves us with a population for the next generation. This is done iteratively until a certain number of generations has been evolved, or when a certain fitness level has been reached, after which the search is terminated. While many of the individuals in the initial population might not do anything

[4] We did try "simpler" search techniques, such as manual trial-and-error or automatic hill-climbing with random restarts and momentum, but the end result was significantly inferior. We believe GA offers a good trade-off between simplicity and effectiveness in this context.

useful, the evolutionary nature of GAs allow some of them to evolve into meaningful, high-performing solutions, and shed the rest in the process.

In our GA, each individual in the population stores rewards for each of the eight actions that can be performed by the scheduler. Initially, these rewards are randomly generated. We evaluate our initial population by conducting execution-driven simulations with each individual's memory scheduler configuration, using a small subset of our application set[5] and determining the fitness of each individual. The fitness-based selection criteria that we use is *tournament selection* combined with *elitist selection* [51, 39]. To perform crossover, we randomly pick two individuals and swap the reward values of an action. Mutation is performed by randomly replacing the reward of an action with another value. Multiple-point crossover and mutations are performed in our experiments, which means that reward values can be swapped or replaced multiple times within a given individual. Once we have the population set for the next generation, it is evaluated against the fitness criteria, and this iterative evolutionary search process continues until we reach 50 generations, at the end of which we are left with a set of rewards, one per possible action, which together constitute our reward function.

In theory, it should be possible to periodically re-calibrate the reward values as the application goes through different execution phases. These rewards would need to be re-learned on the fly by the hardware. We are currently investigating this aspect, but in this paper we confine our solution to static rewards learned offline, which still yields good results and simplifies the design of the

[5]The applications that we use for training are *fft*, *mg*, and *radix* (Section 3.7.3). We picked these because they are the fastest to simulate among the parallel applications that we evaluate. By using a small subset, picked *not* based on behavior but simply on execution time, we speed up training and at the same time minimize the chance of overfitting the final solution to our application set.

scheduler (as we will see, the reward structure is just a small table).

State Attributes

Every memory cycle, the scheduler senses the environment's state via a set of attributes. During the design of the scheduler, it is important to pick the right kind of state attributes that will adequately represent the system environment. There are many candidate attributes that can be used to describe the state of the system. However, to obtain an implementation with reasonable delay and silicon area, it is critical to use a good selection mechanism that picks the right (small) set of state attributes.

Multi-factor Feature Selection

A quick and relatively simple way to accomplish this is to use a linear feature selection process [32]. The designer picks a set of N candidate attributes based on expert intuition. The first step involves simulating N schedulers, each of which uses only one of the N candidate attributes to determine the state of the memory system. Among these N attributes, the designer picks the attribute t_1 that optimizes an objective function (e.g., performance). Then, the designer repeats the selection process with $N - 1$ schedulers, each one considering t_1 and one of the remaining $N - 1$ attributes. After $i \ll N$ iterations, the process concludes, and the i attributes picked determine the state representation.

This linear procedure ignores potentially important interactions between attributes (e.g., attribute t_x alone yields the highest-performing scheduler during

iteration 1, but combination $< t_y, t_z >$ may be superior than $< t_x, t_k >$ for any $1 \leq k \neq x \leq N$), which we experimentally observed are important in our context. In this paper we propose a *multi-factor* approach that takes into account first-order attribute interactions.[6] At the end of the first iteration, we pick the top *two* attributes, and explore the resulting two branches concurrently; at the end of the second iteration, we again pick the top two attributes from each of the two branches, and proceed down four branches; etc. The obvious downside of this approach is that the number of simulations is much higher: for $N = 50$ and $i = 6$, our methodology yields on the order of 8,600 simulations, using the same three training applications per design point. Fortunately, feature selection is a one-time effort made at design time.[7] The resulting attributes are, in principle, inextricably linked to the objective function targeted in the simulations. In Section 2.4, however, we will show that a carefully-trained design can successfully tackle variations of an objective function, by simply reprogramming the reward structure.

Finally, the astute reader will notice that there is a "circular dependence" between automatic feature selection and automatic reward structure derivation: both search a space of completely specified memory scheduler designs. What we do in our paper is to impose a basic ad hoc reward structure during feature selection (Read = Write = 1, rest = 0), but still use the appropriate objective function when evaluating candidate state attributes, and then use the resulting state attributes in the computation of the true reward structure. One could conceive iterating over these two steps to potentially refine the outcome, however for simplicity we do not explore this in this work.

[6]Our mechanism trivially generalizes to higher orders, however we experimented with second-order interactions as well and found no differences in the final state representation.

[7]For each design, we were able to complete all 8,600 simulations in one day.

2.3.2 Implementation

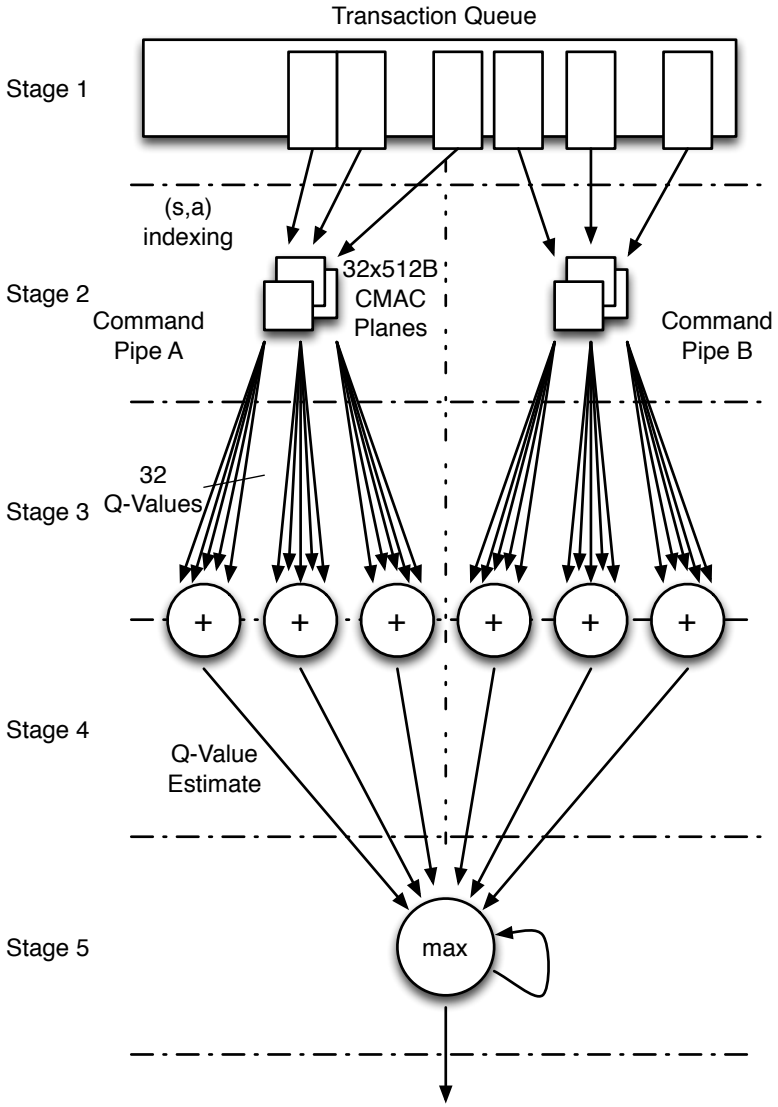

Figure 2.2: Snapshot of the dual five-stage pipeline used in this study to pick the DRAM command with the highest Q-value, among up to 24 eligible actions (four waves of six actions) which can be evaluated every DRAM cycle.

The basic structure of our implementation is necessarily similar to the one described by İpek et al. [32]. In general, only a fraction of the (maximum) 64 outstanding memory requests will have an associated ready DRAM command at each point in time (i.e., one that can be issued to process a memory request

without violating timing constraints). We empirically determine that evaluating 24 ready DRAM commands per DRAM cycle is sufficient to deliver performance that is very close to peak.

For a DRAM scheduler pipeline clocked at 4.27 GHz (same as the CPU, since the memory scheduler sits on chip)[8], controlling a DDR3-1066 system, the Q-value estimation pipeline can be clocked eight times every DRAM cycle. Thus, we use two five-stage command pipes, each capable of considering three ready DRAM commands every processor cycle, for a total of up to 24 ready DRAM commands (four waves of six commands) every DRAM cycle. If there are more memory requests with ready DRAM commands on the queue, the scheduler simply takes the ready DRAM command with the highest Q-value found by the end of the DRAM cycle. (Alternatively, more commands can be considered per cycle if the more pipes are added.) Figure 2.2 shows the five-stage pipeline structure that is used to calculate the Q-values of the proposed scheduler.

In the first stage of the pipeline, the scheduler retrieves six ready DRAM requests (three per command pipe), and it generates the corresponding state-action pairs. (Information about state attributes is actually sensed during the previous DRAM cycle.) In the second pipeline stage, the indices for the Q-value tables are generated and are use to read the Q-values out of the CMAC arrays. The Q-value tables follow a CMAC organization [66]: each Q-value is in fact the result of indexing multiple relatively small tables, each index shifted by an amount predetermined randomly at design time, then adding the result from all such tables together. This structure provides a good balance between resolution and generalization [32]. In our design, we carefully account for the delay and

[8]The clock frequency of IBM's server class Power7 CPU is 4.25 GHz at 45 nm (we target 32 nm in our calculations), with four processor cores and all memory controllers simultaneously on.

power incurred by this structure, which essentially amounts to 32 SRAM tables, each with 256 two-byte (fixed-point) entries, three read ports and one write port, and where the updates to the Q-values are done using fixed point arithmetic. The index for the CMACs are generated by concatenating the higher order bits of the state attributes. This concatenated value is then XOR-ed with a constant, depending on the corresponding action that was chosen. Finally, the XOR-ed value is passed through a hash function (to reduce storage requirements) to get the index into the Q-value tables. The next two stages are used to add up the 32 Q-values of the three different commands analyzed per command pipe, which have been read out from the CMAC arrays. In the fifth and final stage of the pipeline, the maximum Q-value seen so far is compared against the six new Q-values and updated as needed.

2.4 Reconfigurability

We have described our general framework as a design-time mechanism to build self-optimizing memory schedulers. In this section, we briefly outline how one can in fact implement a self-optimizing scheduler that can accommodate multiple objective functions, and target each as desired on the field (post-silicon), either at boot time or even dynamically by the operating system.

Actions – The actions available to the self-optimizing scheduler are simply the possible DRAM commands, irrespective of the objective function. Therefore, no hardware changes are required to map actions to commands for a different configuration.

Rewards – There are as many rewards as there are actions in the RL agent. These rewards can be stored easily using a table inside the RL agent. There are two approaches to storing rewards for multiple configurations: (a) A single table that can be programmed as needed (e.g., by the operating system) to store the values relevant to the desired objective function. (b) The memory scheduler can store a small number of pre-programmed tables, and use the appropriate one as needed. A combination of both is also possible (i.e., multiple programmable tables).

State Attributes – Although selecting state attributes is an offline process, the resulting state attributes must be sensed by adding the appropriate hardware (e.g., a counter, a read port, etc.). However small this may be, as the number of metrics of interest increase (e.g., to support more than one possible objective function), so will the aggregate hardware overhead. In general, the solution will be a compromise in the number of observable state attributes, driven by potential gains vs. area and complexity.

Fortunately, the CMAC structure that stores the Q-values (the main storage overhead) is itself attribute-agnostic–the differentiation across objective function resides in what attributes are actually sensed, but the indexing into the CMAC is identical regardless of the attribute [32]. Thus, we can easily add multiplexers to steer the right set of attributes to the CMAC depending on the target objective function.

2.5 Experimental Methodology

2.5.1 Architecture Model

The baseline processor model integrates eight cores and supports a DDR3-1066 memory subsystem with four independent, address-interleaved memory channels. Our memory subsystem model (DIMM structure, timing, and power) follows the Micron DDR3 DRAM specification [5, 7, 8], including refresh. The micro-architectural features of the baseline processor are shown in Table 3.2; the parameters of the L2 cache, the memory system, and the DDR3 SDRAM power model are shown in Tables 3.3 and 3.5. We implement our model by extending the SESC simulation environment appropriately [58].

We compute the energy overhead of the self-optimizing scheduler designs as follows, assuming a 32 nm process except where noted.

Dynamic power – Q-value computation: Computing a Q-value consists of three basic steps: (a) generating the array indices, (b) reading the Q-values, and (c) adding the Q-values and determining the maximum Q-value. To generate the array indices, we first read the six selected state attributes and concatenate the higher order bits. This is then XOR-ed with a random number and passed through a hash function. Reading the state attributes and indexing into a hash function can be approximated as a dynamic SRAM read each and consumes 1.4 pJ per read (from CACTI 6.5 [1]). The XOR function takes 0.23 pJ (an XOR function is conservatively approximated to consume the same power as an adder implemented in an older 70 nm technology [35]). We use CACTI to estimate the energy expended in reading out the Q values from the SRAM arrays. Each

SRAM read consumes 0.78 pJ, and the total dynamic SRAM energy for reading out the Q values from the 32 matrices per command is 24.96 pJ. We estimate the power consumed by the adders that sum up the 32 Q-values to be 1.0 mW each [35], and accordingly calculate the energy consumed by the 16 adders used in each RL pipeline to be 3.75 pJ (adding the 32 Q-values takes up two pipeline cycles). We conservatively assume that the comparator consumes the same power as the adder. Since a maximum of 24 commands can be analyzed every DRAM cycle, a maximum of 24 final Q-values need to be compared each cycle, and the energy estimated to do so is 3 pJ. The total RL pipeline energy consumed is then 32 pJ.

Dynamic power – Q-value update: To update the Q-values using the SARSA update rule we use three multipliers and three adders. The energy consumed to perform this operation is 2.1 pJ [35]. Finally the Q-values need to be written back into the SRAM arrays (32 per command, 64 in all). This consumes 49 pJ as estimated from CACTI.

Leakage: Using CACTI, we also estimate the total leakage power per CMAC matrix to be 0.36 mW, and consequently the leakage energy to be 5.4 pJ per DRAM cycle.

Overall, we find the energy overhead of the self-optimizing schedulers to be negligible (the equivalent of about 2% of the energy consumed by the DRAM on average). Nevertheless, the energy and Et^2 results reported in sections 2.6.1, 2.7.1, and 2.8 *do* include this overhead. Moreover, in our results we effectively assess zero energy overhead for the competing FR-FCFS and Pwr-FR-FCFS schemes.

2.5.2 Applications

We evaluate our proposed MORSE scheduler on a wide variety of parallel and multi-programmed workloads from the server and desktop computing domains. We simulate nine memory-intensive parallel applications, running eight threads each, to completion. Our parallel workloads constitute a good mix of scalable scientific programs from different benchmark suites, as shown in Table 3.6. For our multi-programmed workloads, we use 14 four-application bundles from the SPEC 2000 and NAS benchmark suites, which constitute a healthy mix of CPU-, cache-, and memory-sensitive applications. Table 2.5 describes the bundles. In each case, we fast-forward each application for half a billion instructions, and then execute the bundle concurrently until *all* applications in the bundle have committed at least half a billion more instructions.

2.6 Case I: Performance

In this section we use our general framework to design a performance-oriented self-optimizing memory scheduler, and provide quantitative evidence of its superiority with respect to İpek et al.'s original design [32].

İpek et al.'s scheduler uses a simple ad hoc reward structure, which assigns a reward of 1 for reads and writes (immediately "productive" actions), and 0 otherwise. To allow the controller to use the most appropriate set of state attributes for our experimental setup (different from theirs), we re-run their proposed linear feature selection [32], using their six winning attributes, plus another 44 relevant ones that we come up with. By using their original attributes as part of

Table 2.1: Core Parameters.

Technology	32 nm
Frequency	4.27 GHz
Number of cores	8
Fetch/issue/commit width	4/4/4
Int/FP/Ld/St/Br Units	2/2/2/2/2
Int/FP Multipliers	1/1
Int/FP issue queue size	32/32 entries
ROB (reorder buffer) entries	96
Int/FP registers	96 / 96
Ld/St queue entries	24/24
Max. unresolved br.	24
Br. mispred. penalty	9 cycles min.
Br. predictor	Alpha 21264 (tournament)
RAS entries	32
BTB size	512 entries, direct-mapped
iL1/dL1 size	32 kB
iL1/dL1 block size	32 B/32 B
iL1/dL1 round-trip latency	2/3 cycles (uncontended)
iL1/dL1 ports	1 / 2
iL1/dL1 MSHR entries	16/16
iL1/dL1 associativity	direct-mapped/4-way
Memory Disambiguation	Perfect
Coherence protocol	MESI
Consistency model	Release consistency
RL discount rate parameter γ	0.95
RL learning rate parameter α	0.1

the set, we make sure the resulting scheduler is *at least* as good as the original one. In our experiments, we call this configuration *Ipek*.

MORSE-P is our performance-oriented self-optimizing scheduler. It is de-

Table 2.2: Parameters of the shared L2 and DRAM.

Shared L2 Cache Subsystem	
Shared L2 Cache	4 MB, 64 B block, 8-way
L2 MSHR entries	64
L2 round-trip latency	32 cycles (uncontended)
Write buffer	64 entries
Micro DDR3-1066 DRAM [5]	
Transaction Queue	64 entries
Peak Data Rate	6.4 GB/s
DRAM bus frequency	533 MHz (DDR)
Number of Channels	4
DIMM Configuration	Quad rank
Number of Banks	8 per rank
Row Buffer Size	1 KB
Address Mapping	Page Interleaving
Row Policy	Open Page
Burst Length	8
tRCD	7 DRAM cycles
tCL	7 DRAM cycles
tWL	6 DRAM cycles
tCCD	4 DRAM cycles
tWTR	4 DRAM cycles
tWR	8 DRAM cycles
tRTP	4 DRAM cycles
tRP	7 DRAM cycles
tRRD	4 DRAM cycles
tRTRS	2 DRAM cycles
tRAS	20 DRAM cycles
tRC	27 DRAM cycles
Refresh Cycle	8,192 refresh commands every 64 ms
tRFC	59 DRAM cycles

Table 2.3: Parameters of the Micron DDR3-1066 DRAM power management features [5, 8, 7].

IDD0	90 mA
IDD3PF	55 mA
IDD3PS	55 mA
IDD2PF	35 mA
IDD2PS	12 mA
IDD2N	70 mA
IDD3N	80 mA
IDD4R	200 mA
IDD4W	255 mA
tFAW	20 DRAM cycles
tACTPDEN	1 DRAM cycles
tPREPDEN	1 DRAM cycles
tRDPDEN	12 DRAM cycles
tWRPDEN	18 DRAM cycles
tXP	4 DRAM cycles
tXPDLL	13 DRAM cycles
Vdd	1.8V

rived using our proposed automatic reward derivation and multi-factor feature selection procedures, using performance as the objective function and the same 50 candidate state attributes.

The state attributes selected via multi-factor feature selection for MORSE-P are: (1) Number of reads (load/store misses) in the transaction queue. (2) Number of writes in the transaction queue. (3) Number of reads for the current rank under consideration. (4) If the memory request is related to a load miss, the order of the load relative to the other loads in the transaction queue for the corresponding core. (5) Number of writes in the transaction queue that reference rows that are open. (6) Number of commands for the rank under consideration

Table 2.4: Simulated parallel applications and their input sets.

	Data Mining	[57]
scalparc	Decision Tree	125k pts., 32 attributes
	NAS OpenMP	[12]
mg	Multigrid Solver	Class A
cg	Conjugate Gradient	Class A
	SPEC OpenMP	[11]
swim-omp	Shallow water model	MinneSpec-Large
equake-omp	Earthquake model	MinneSpec-Large
art-omp	Self-organizing Map	MinneSpec-Large
	Splash-2	[73]
ocean	Ocean movements	514×514 ocean
fft	Fast Fourier transform	1M points
radix	Integer radix sort	2M integers

when it is powered down.

Attributes (1) and (2) help the scheduler determine how to balance reads and writes in the transaction queue ; (3) prioritizes among reads from different ranks ; (4) is used to prioritize among load misses from the same core ; (5) helps the RL scheduler issue writes in bursts so as to better manage write-to-read delays ; and finally (6) determines if it is time to power up ranks if there are memory requests waiting to access the rank.

The rewards obtained from the GA-based automatic reward derivation process are: Activate = 1.59, Precharge = -1.47, Preemptive Precharge = -1.47, Read(Load) = 2.00, Read(Store) = 1.59, Write = 0.88, PwDn = -0.27, PwUp = 0.30, NoOp = 0.58.[9] The resulting values are very interesting. On the one hand, in

[9]We arbitrarily set up the reward structure to use higher (lower) values as positive (negative) rewards, which is typical in machine learning texts.

Table 2.5: Multiprogrammed Configurations evaluated. C, P, and M stand for Cache-, Processor-, and Memory-sensitive, respectively [12, 28].

Art Mcf Ep Vpr (TFEV)	M M M C
Mg Sp Mesa Vpr (GPMV)	M P P C
Mg Cg Apsi Crafty (GCAY)	M M P C
Apsi Art Mg Swim (ATGS)	P M M M
Mg Cg Mesa Mgrid (GCMD)	M M P M
Art Is Vpr Parser (TIVR)	M M C C
Mg Cg Vpr Swim (GCVS)	M M C M
Cg Mesa Is Crafty (CMIY)	M P M C
Twolf Cg Mesa Is (OCMI)	C M P M
Cg Lu Sp Art (CLPT)	M P P M
Art Is Vpr Wupwise (TIVW)	M M C C
Cg Vpr Wupwise Mesa (CVWM)	M C C P
Twolf Cg Mesa Parser (OCMR)	C M P C
Lu Cg Mesa Is (LCMI)	P M P M

many cases their magnitude relative to each other makes intuitive sense: For example, reads and writes have a high positive reward. Precharge is negative while Activate is positive, which hints at the importance of exploiting row buffer locality. PwDn is negative, as the penalty to bring up a rank once it has been powered down is 4-13 cycles.

On the other hand, other aspects of this reward assignment are not so obvious. For example, the specific ratios among the reward values for the different actions are non-intuitive. It is intriguing that, despite the fact that the objective function is straight performance, a PwDn-PwUp action sequence yields a slightly positive aggregate reward (-0.27+0.3), even though powering up a dormant rank incurs a penalty of 4-13 cycles. Note also that NoOp (which competes

with PwUp when a rank is powered down) is assigned a definitely positive reward, even though keeping a rank powered down does not directly benefit performance. We will revisit this later.

2.6.1 Evaluation

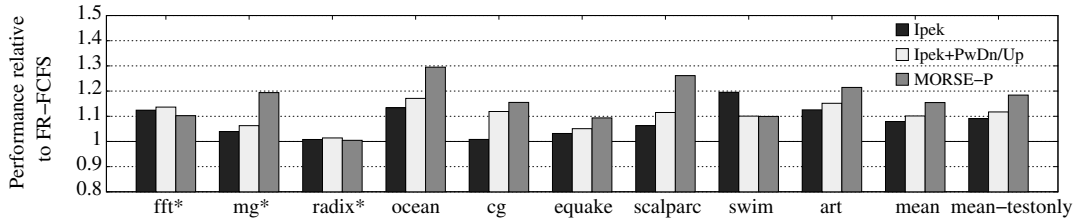

Figure 2.3: Performance (higher is better) of Ipek, Ipek+PwDn/Up and MORSE-P, normalized to that of FR-FCFS. The three applications used during training are marked with an asterisk; mean-testonly excludes them.

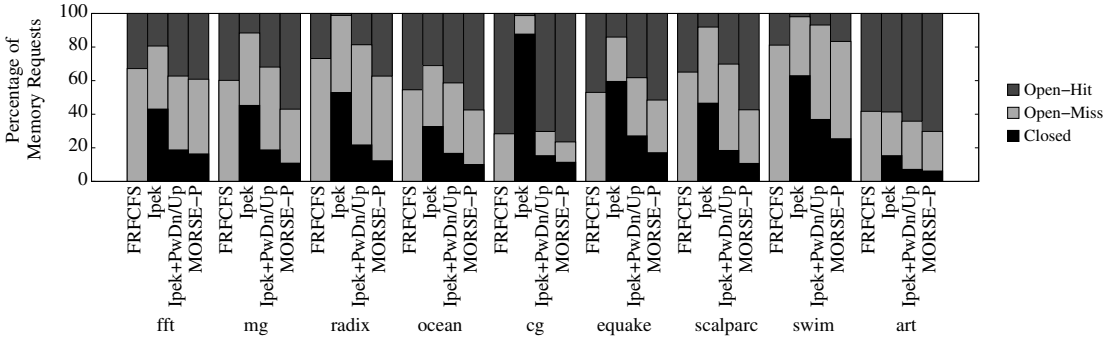

Figure 2.4: Breakdown of expected page status at the time a new memory request for that page arrives for FR-FCFS, Ipek, Ipek+PwDn/Up and MORSE-P.

Figure 2.3 shows performance data for all the configurations studied. The proposed MORSE-P scheduler has a speedup of 15.5% (18.4% excluding the three applications used during training) and 7% (8.6%) with respect to FR-FCFS

and Ipek, respectively. It would seem that the additional sophistication in picking state attributes and immediate reward values does pay off. We now try to understand how MORSE-P achieves its superiority.

Figure 2.4 breaks down the expectation of page status for a newly arrived command: *Open-Hit* if the page is already open in the corresponding bank; *Open-Miss* if there's an open page which is not the right one (which will impose a Precharge-Activate sequence to get the right page); and *Closed* if there's no open page at that bank–i.e., the scheduler has preemptively precharged a page, probably in an attempt to save time the next time a page needs to be activated. Evidently, FR-FCFS, which is an open-page algorithm, simply does not expect to find a closed page in steady state. On the other hand, the self-optimizing configurations Ipek and MORSE-P do precharge pages proactively if they predict a long-term benefit. The plot shows that MORSE-P is in fact superior to FR-FCFS and Ipek, in that the expectation of Open-Hit is highest almost universally. This increased hit rate correlates well with bottom-line performance in many applications.

Interestingly, the expectation of finding a bank closed in Ipek is significantly higher than in MORSE-P. It turns out that this is in part a potentially undesirable side effect of İpek et al.'s imposition that a NoOp be allowable only if no legal commands (in particular, Precharge) exist, which was put in place to speed up convergence [32]. This means that Ipek is hardwired to closing pages (instead of doing nothing) when there are no other options available, regardless of the long-term benefit of doing so. In the case of MORSE-P, the scheduler learns by itself that it may "bypass" this restriction, by judiciously exploiting the PwDn/PwUp actions available in the DDR3 interface (i.e., it can force NoOp to be a legal

choice indirectly by powering down a rank, effectively making Precharge to any bank within that rank ineligible until the rank is powered back up). As indicated before, the reward values obtained by our GA-based procedure hint at the potential benefit of this subterfuge, as it assigns a slightly positive aggregate reward to PwDn+PwUp, and a definitely positive reward to NoOp.

Thus, it is conceivable that Ipek might also learn to do the same if PwDn/PwUp actions are made available, potentially closing the performance gap with MORSE-P. This is precisely what the Ipek+PwDn/Up configuration in the plots tries to answer. In that configuration, PwDn/PwUp are available actions with an immediate reward of 0 (consistent with the ad hoc reward function employed), and linear feature selection is re-run. Figures 2.3 and 2.4 show the performance and expected page status for the Ipek+PwDn/Up configuration. As we can see, the expectation of finding a bank closed drops dramatically to levels similar to those in MORSE-P. When looking at overall performance, however, not much is gained on average, and a significant gap remains, which means that appropriate state attribute and reward values are, in fact, the primary contributors to performance in MORSE-P vs. Ipek.

Looking at individual applications, however, we see a couple of outliers. In *cg*, *both* Ipek+PwDn/Up and MORSE-P derive noticeable benefit from having PwDn/PwUp available, even if MORSE-P still edges out Ipek+PwDn/Up significantly (Figure 2.3). *cg* strongly favors an open page policy, as evidenced by the large expectation of a page hit for FR-FCFS (which is open-page). Ipek destroys such potential by being forced by design to close pages prematurely over doing nothing (Figure 2.4). On the other hand, *swim* seldom hits on open pages for FR-FCFS, and actually performs best with Ipek, where the NoOp restriction

is unavoidable and it results in a high number of preemptive precharges. Relaxing this constraint actually hurts Ipek+PwDn/Up and MORSE-P virtually equally with respect to Ipek, although they still outperform full-open-page FR-FCFS significantly. In other words, while these configurations do learn to preemptively close pages and derive speedups as a result, in the case of *swim* they fall short of the level of aggression with which they could apply it, as (accidentally) evidenced by Ipek. (Note that *swim* is not part of our training set. While the obvious temptation is to include it, this would amount to overfitting.)

2.7 Case II: Energy Efficiency

In this section, we use our general framework to design a self-optimizing memory controller that will strive to optimize for energy-delay-squared (Et^2)–a metric that combines performance and energy consumption. Our evaluation provides quantitative evidence that the resulting controller is significantly superior to a power-aware extension of FR-FCFS that includes Hur and Lin's Queue-Aware Power-Down mechanism [30], which we refer to as *Pwr-FR-FCFS*.[10]

In this section, *MORSE-E* is our energy-efficient self-optimizing scheduler, which is naturally derived using our proposed automatic reward derivation and multi-factor feature selection procedures, using Et^2 as the objective function and the same 50 candidate state attributes used in Section 2.6.

The state attributes selected via multi-factor feature selection for MORSE-E are:

[10]We also experimented with Hur and Lin's AHB [29] and found the results to be very similar.

(1) Number of ranks that are idle and powered up. (2) Number of rows that are open with no pending commands for a given rank. (3) Number of commands that will be negatively affected if a Precharge is issued for the corresponding open row (i.e., they would subsequently miss on the page being precharged). (4) Number of writes in the transaction queue. (5) Number of reads for the current rank under consideration. (6) If the memory request is related to a load miss, the order of the load relative to the other loads in the transaction queue for the corresponding core.

Attributes 1 and 2 address power, while the rest are geared primarily towards performance. This is good news, as energy efficiency targets both these metrics. Specifically, those two attributes help the scheduler determine when it is time to power down ranks. Attribute 3 helps maintain row-buffer locality, by determining the number of commands that get affected by a precharge. Attribute 4 helps keep a balance between reads and writes in the transaction queue. Attribute 5 prioritizes among reads from different ranks. Lastly, Attribute 6 is used to prioritize among load misses from the same core.

The rewards obtained from the GA-based automatic reward derivation process are: Activate = -0.21, Precharge = -3.52, Premptive Precharge = -2.06, Read(Load) = 3.71, Read(Store) = 0.86, Write = 1.38, PwDn = -2.06, PwUp = -3.10, NoOp = -1.78. As before, reads and writes always have higher positive reward (better performance) than the other actions. This time, PwUp carries a negative reward–even more so that PwDn. This makes sense, as once the scheduler has decided to power down a rank, it should be because it intends to keep it that way for a while and save energy. Still, as in the case of the straight-performance scheduler, a Precharge is more negative than a PwDn-PwUp sequence, which

helps preserve row-buffer locality.

2.7.1 Evaluation

Energy-Delay Squared

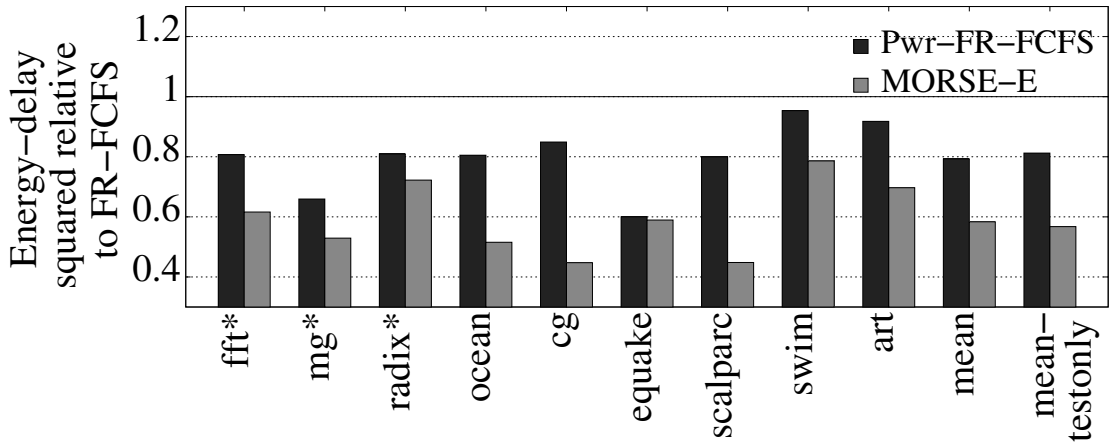

Figure 2.5: Energy-delay squared Et^2 (lower is better) for the energy-aware configurations considered in this study, normalized to that of FR-FCFS. The three applications used during training are marked with an asterisk; mean-testonly excludes them.

Figure 2.5 compares the configurations considered in this study in terms of Et^2, normalized to that of FR-FCFS (which is not energy-aware). Our proposed MORSE-E DRAM scheduler reduces Et^2 by 42% (43% excluding the three applications used during training) when comapred to FR-FCFS, and by 26% (30%) when compared to Pwr-FR-FCFS. MORSE-E outperforms Pwr-FR-FCFS by 18% or more for all applications except *radix* (11.0%) and *equake* (2.0%) .

Performance

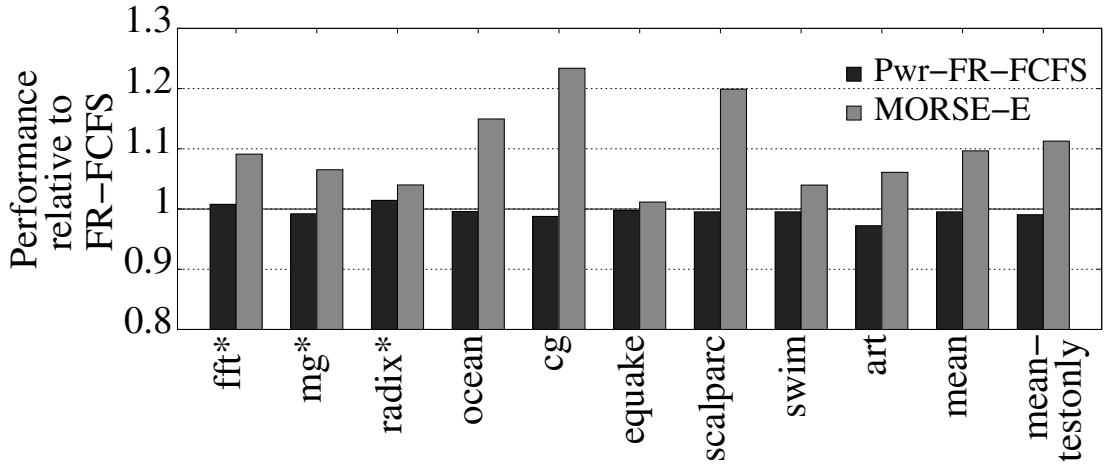

Figure 2.6: Performance (higher is better) of the energy-aware configurations considered in this study, normalized to that of FR-FCFS. The three applications used during training are marked with an asterisk; mean-testonly excludes them.

Figure 2.6 shows performance data for all the configurations studied. The proposed MORSE-E scheduler has a speedup of 9.7% (11%) and 10% (11%) with respect to FR-FCFS and Pwr-FR-FCFS respectively. This is very good news: The proposed scheme not only beats Pwr-FR-FCFS handsomely in energy efficiency, it does so while actually delivering a performance gain.

Energy

Figure 2.7 shows the energy consumed in executing the various applications. Naturally, the energy-oblivious configuration (FR-FCFS) has the highest energy consumption. Our proposed MORSE-E scheduler yields energy savings of 20% (30% excluding the training apps) and 11% (12%) on average over FR-FCFS and

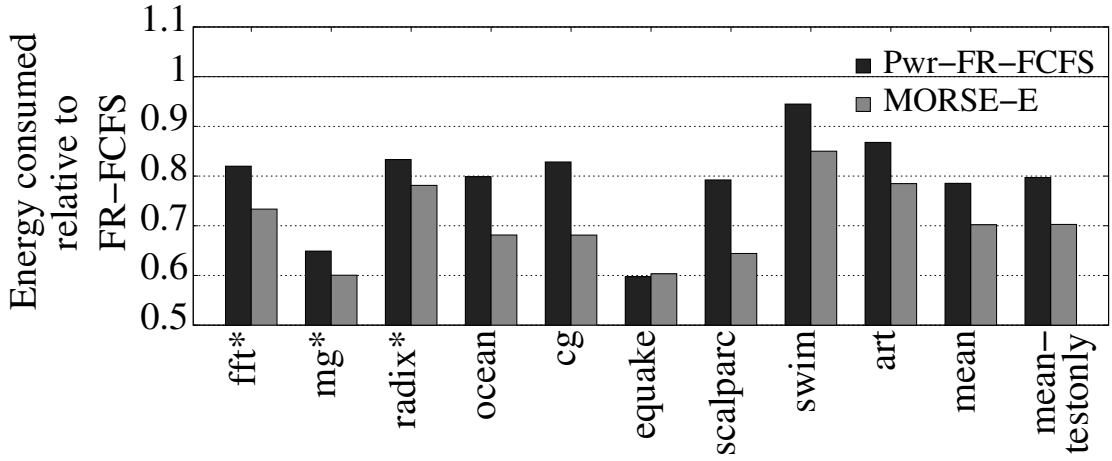

Figure 2.7: Energy (lower is better) consumed by the energy-aware configurations considered in this study, normalized to that of FR-FCFS. The three applications used during training are marked with an asterisk; mean-testonly excludes them.

Pwr-FR-FCFS, respectively.

2.7.2 Analysis

Number of Active Ranks

Figure 2.8 shows the DRAM transaction queue occupancy and active ranks, averaged over intervals of 5,000 DRAM cycles, for the NAS-OpenMP application *mg* (this behavior is representative of most of the applications studied) using Pwr-FR-FCFS. From the plot, we can see that, throughout the entire execution cycle of the application, there are relatively few instances where the DRAM queue occupancy exceeds the number of active ranks in the memory system. This is consistent with our expectation for high-end servers, where peak de-

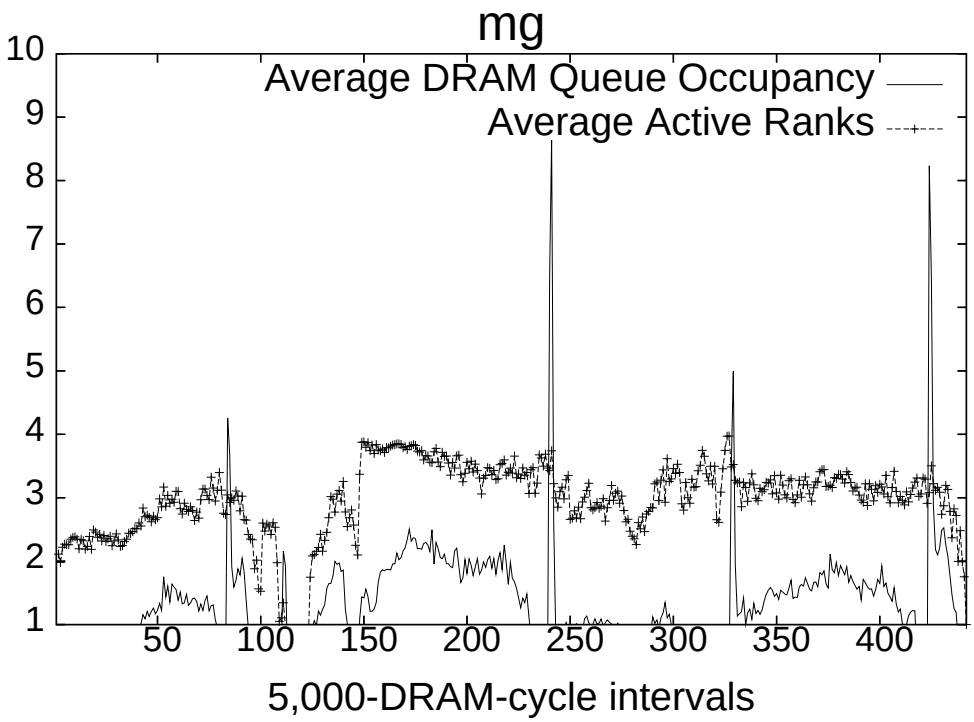

Figure 2.8: DRAM transaction queue occupancy and average number of active ranks per channel, averaged over 5,000-DRAM-cycle intervals, for the *mg* application in Pwr-FR-FCFS. (The behavior is representative of the other applications.)

mand must be served efficiently to ultimately deliver good performance. Thus, it is extremely important to have an efficient power management scheme that puts idle devices into low-power states and activates them at the right time to avoid significant losses in performance. Figure 2.9, which plots the average number of active ranks per channel, shows how MORSE-E is able to reduce the average number of active DRAM devices significantly over Pwr-FR-FCFS.

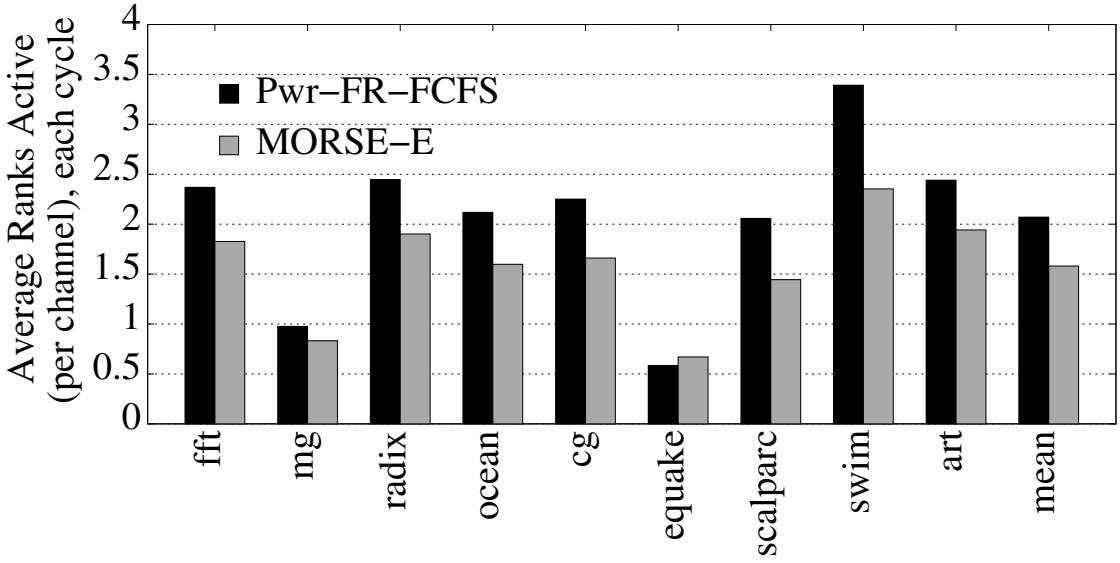

Figure 2.9: Average Number of DRAM ranks (per channel) that are powered up (active) each cycle in Pwr-FR-FCFS and MORSE-E.

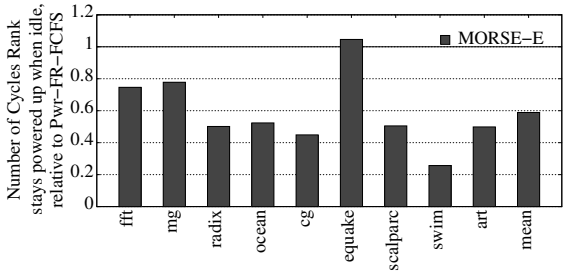

Figure 2.10: Number of cycles DRAM ranks stay powered up when no outstanding command exists in the DRAM transaction queue for the corresponding rank for the MORSE-E configuration, normalized to Pwr-FR-FCFS.

Impact of the Selected Attributes on Energy Efficiency

The energy-aware feature selection process in MORSE-E picked attributes that hinted at situations that help improve DRAM background power consumption. In this section, we provide insights into the benefits of picking those attributes,

and how they help improve energy efficiency. Figure 2.10 shows the number of DRAM cycles a rank remains powered up when no outstanding command is present in the transaction queue for MORSE-E and Pwr-FR-FCFS. Recall that one of the states picked by the feature selection process evaluates the ranks that are idle and powered up (Attribute 1). MORSE-E is able to proactively determine when a rank can be placed in low power mode without hurting pending requests. As a result, ranks stay in this mode for longer on average, resulting in greater energy efficiency.

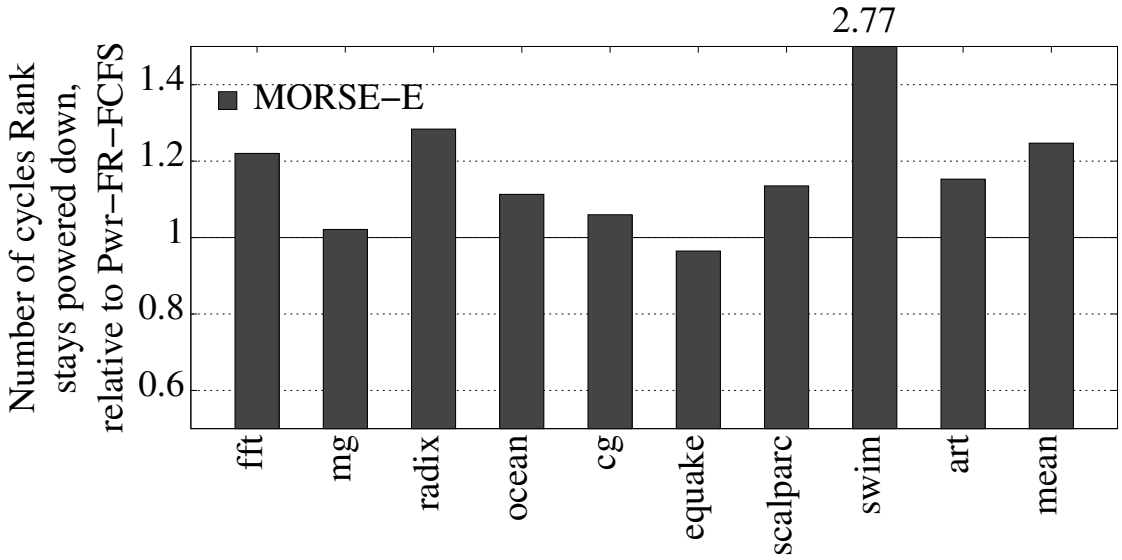

Figure 2.11: Number of cycles DRAM ranks stay powered down for the MORSE-E configuration, normalized to Pwr-FR-FCFS.

Figure 2.11 shows the number of DRAM cycles a rank remains powered down in MORSE-E and in Pwr-FR-FCFS normalized to that of Pwr-FR-FCFS. Powering up a rank from low power mode takes 4-13 cycles, and hence, if done prematurely, will lead to increase in energy consumption, while if done later, will lead to a loss in performance. The MORSE-E scheduler is able to learn this

fine balance based on the state information and the reward function.

2.7.3 Effect on Multiprogrammed Workloads

We now assess the robustness of this scheduler by evaluating it in a scenario different from the one used in the design phase, namely multiprogrammed workloads.

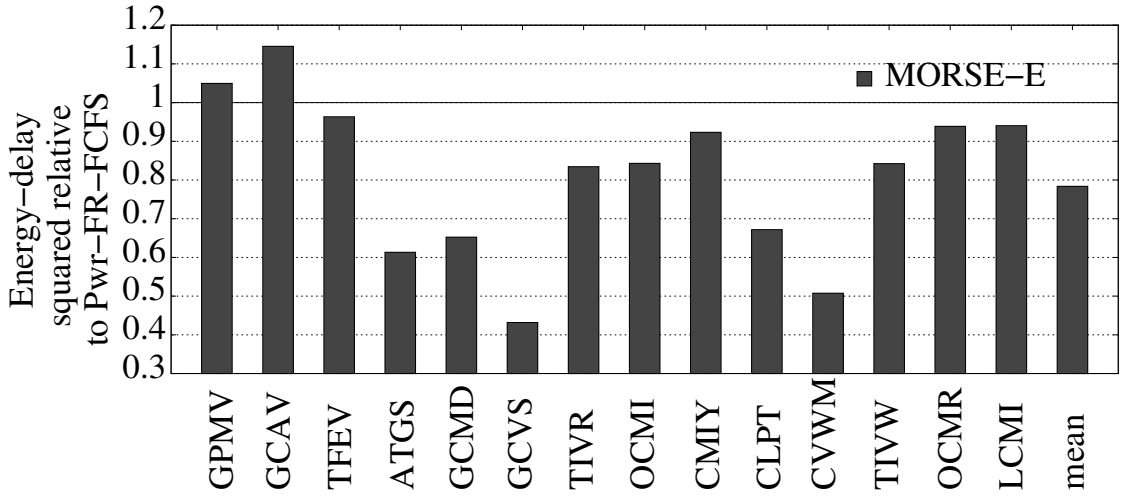

Figure 2.12: Energy-delay squared Et^2 (lower is better) for the MORSE-E configuration, normalized to that of Pwr-FR-FCFS.

Figures 2.12, 2.13, and 2.14 shows Et^2, energy, and performance of MORSE-E relative to that of Pwr-FR-FCFS in each case. (To compute performance and Et^2, we measure the time needed in each case to execute *at least* half a billion instructions in *each* of the workloads after warm-up.) The results are very reassuring: Even though MORSE-E has been trained on a very small number of parallel applications and *no* multiprogrammed workloads whatsoever, our methodology has been able to produce a memory scheduler that is relatively robust in this

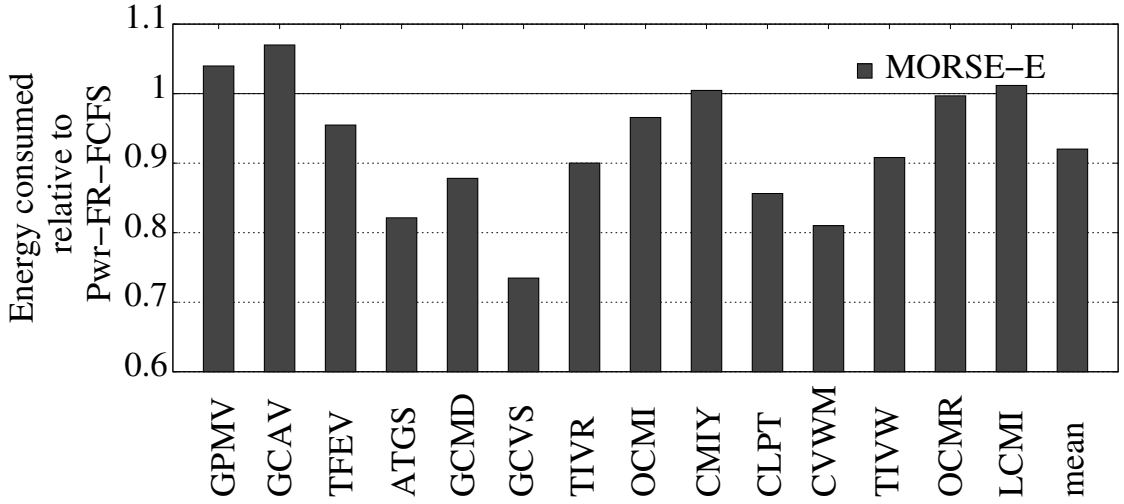

Figure 2.13: Energy consumption (lower is better) for the MORSE-E configuration, normalized to that of Pwr-FR-FCFS.

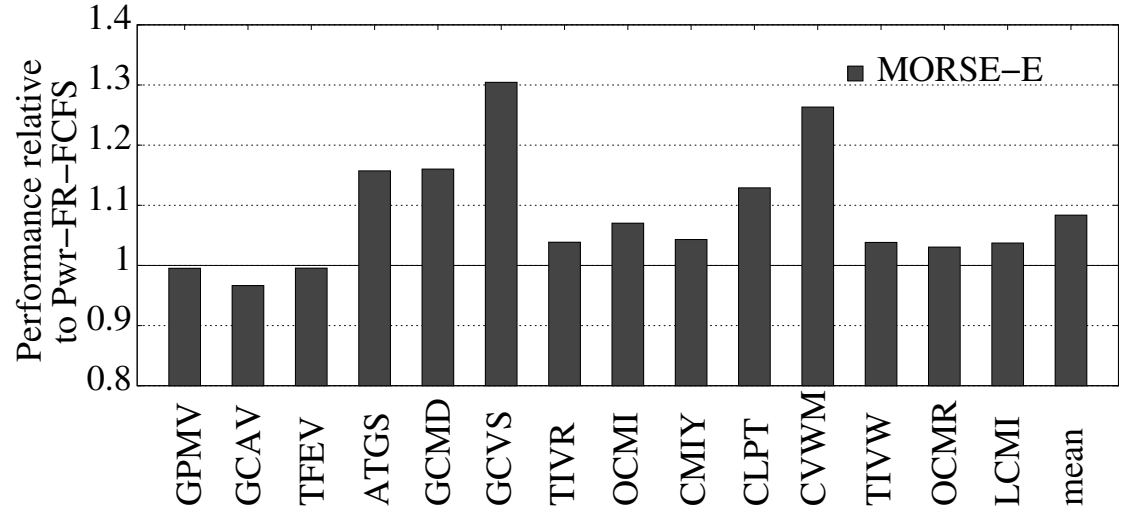

Figure 2.14: Performance (higher is better) for the MORSE-E configuration, normalized to that of Pwr-FR-FCFS.

different context. Specifically, MORSE-E improves Et^2 by 22% on average over Pwr-FR-FCFS, which results from simultaneously saving 8% energy on average and delivering a 8.3% average speedup. Only two workloads experience an Et^2 degradation (below 15% in any case), most likely stemming from the fact that

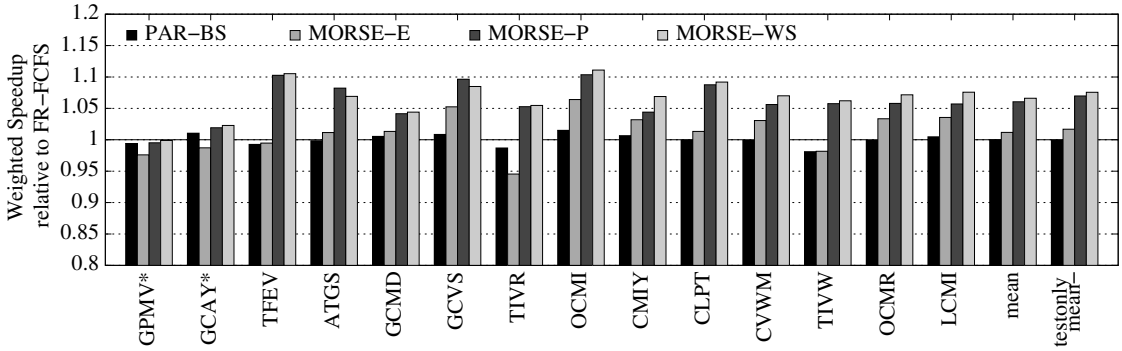

Figure 2.15: Weighted speedup (higher is better), normalized to that of FR-FCFS. The two workloads used during training for MORSE-WS are marked with an asterisk; mean-testonly excludes them.

MORSE-E is not specifically designed for multiprogrammed workloads.

2.8 Case III: Throughput/Fairness

In this section, we describe a self-optimizing scheduler design that targets system throughput/fairness for multiprogrammed workloads. We use weighted speedup as the objective function. It is the sum of the ratio, for each application, of the IPC obtained in the multiprogrammed scenario over the IPC that the application would enjoy if it were to run alone in the same system [62]. We also conducted experiments using harmonic speedups [46], but obtained virtually identical results.

MORSE-WS is a configuration for which feature selection and GA-based rewards derivation have been conducted, using weighted speedup as the objective function. The two workloads used for training are *GPMV* and *GCAY* (Section 3.7.2). A key detail to notice is that, while weighted speedup would be complex to observe directly on the field, in our framework it is easy to target at

design time through simulations. Once features and rewards have been tuned for that metric, they will "embed" it behaviorally on the field, without ever needing to actually measure it.

We also report the performance of plain MORSE-P, and MORSE-E, which represent cases where no post-silicon changes are possible. Finally, we also evaluate a state-of-the-art scheduler form the literature that also targets weighted speedup (PAR-BS) [53].

Figure 2.15 shows the weighted speedups obtained for all configurations, relative to FR-FCFS. For our architecture organization and applications (different from those of PAR-BS's original paper [53]), PAR-BS offers negligible benefit over FR-FCFS. MORSE-WS, on the other hand, significantly outperforms PAR-BS. We also notice that MORSE-P matches MORSE-WS's performance, whereas MORSE-E does not. This is maybe not too surprising: Although neither MORSE-P not MORSE-E were designed to target weighted speedup of multiprogrammed workloads, MORSE-P's objective is more closely aligned with MORSE-WS's. MORSE-E's result, on the other hand, is probably representative of a scenario where the fundamental differences between train (-E) and test (-WS) objective functions are more significant, and in that case the bottom line suffers. Still, given the fact that MORSE-P was trained using very different workloads (parallel applications), the results are further testament to MORSE's robustness.

2.9 Related Work

Hur and Lin [30] propose a simple queue-aware power-down policy for exploiting low-power modes of modern DRAMs. In addition, they also propose the use of a power-aware memory scheduler that encodes several scheduling goals in finite state machines (FSM), and chooses among the FSMs using a probabilistic arbiter. The FSMs encode the ratio of reads and writes serviced in the past, groups same rank commands together and groups commands that optimizes for expected latency. However, one drawback of their power-aware memory scheduler is that it does not consider row precharges, row activations, rank power up and rank power down as separate, individual DRAM commands and hence does not address different trade-offs involved in DRAM scheduling.

Lebeck et al. [41] explore the interaction of page placement policies with the power management techniques used in DRAM systems. Their preliminary experiments using offline profiling of memory accesses show that there is potential in employing page placement policies by an informed operating system to complement the hardware power management strategies.

Fan et al. [23] investigate memory controller policies for manipulating DRAM power states in cache-based systems. They develop an analytical model that approximates the idle time of memory devices, so that they can be powered down and powered up accordingly. However, their model does not sufficiently capture changes to workload demands, and does not learn the long-term performance impact of a scheduling decision, both of which are major benefits of our energy-efficient scheduler.

In other related work, Fan et al. [24] show that there is a positive synergistic

effect between DVS and power-aware memories that can transition into low power states, which offers potential for energy savings. Sudan et al. [65] make an interesting observation that a large number of memory accesses mechanisms to heavily accessed OS pages are to a small chunk of contiguous cache blocks. Thus co-locating these chunks from different pages will improve row buffer locality, and energy consumption.

2.10 Conclusions

We have proposed the use of genetic algorithms as a general way to systematically derive reward functions for RL-based DRAM schedulers. We have shown that this mechanism is not only capable of targeting arbitrary objective functions, it yields higher-performing schedulers than previously-proposed self-optimizing solutions that employed an ad hoc reward structure. In the process, we have also proposed a *multi-factor feature selection* procedure for designing self-optimizing schedulers that takes into account first-order interactions among RL state attributes. We use this general mechanism to present three memory scheduler designs that target performance, energy efficiency, and throughput/fairness, respectively. Our results significantly outperform the state of the art in the literature in each case. We also show evidence that our designs are robust across workload classes and objective functions when train and test objective functions are similar enough in nature (e.g., both performance-oriented metrics).

CHAPTER 3

UNDERSTANDING AND MITIGATING REFRESH OVERHEADS IN HIGH DENSITY DDR4 MEMORY

ABSTRACT

Recent DRAM specifications exhibit increasing refresh latencies. A refresh command blocks a full rank, decreasing available parallelism in the memory subsystem significantly, thus decreasing performance. Fine Granularity Refresh (FGR) is a feature recently announced as part of JEDEC's DDR4 DRAM specification that attempts to tackle this problem by creating a range of refresh options that provide a trade-off between refresh latency and frequency.

In this chapter, we first conduct an analysis of DDR4 DRAM's FGR feature, and show that there is no one-size-fits-all option across a variety of applications. We then present *Adaptive Refresh (AR)*, a simple yet effective mechanism that dynamically chooses the best FGR mode for each application and phase within the application.

When looking at the refresh problem more closely, we identify in high-density DRAM systems a phenomenon that we call *command queue seizure*, whereby the memory controller's command queue seizes up temporarily because it is full with commands to a rank that is being refreshed. To attack this problem, we propose two complementary mechanisms called *Delayed Command Expansion (DCE)* and *Preemptive Command Drain (PCD)*.

Our results show that AR does exploit DDR4's FGR effectively. However, once our proposed DCE and PCD mechanisms are added, DDR4's FGR becomes

redundant in most cases, except in a few highly memory-sensitive applications, where the use of AR does provide some additional benefit. In all, our simulations show that the proposed mechanisms yield 8% (14%) mean speedup with respect to traditional refresh, at normal (extended) DRAM operating temperatures, for a set of diverse parallel applications.

3.1 Introduction

The dynamic nature of DRAM requires logic to carefully track the DRAM lines that need to be refreshed, and to issue refresh commands in a timely fashion. For a long time, refresh commands were relatively short and infrequent, and thus system performance and power were not significantly impacted. As DRAM density increased, industry decided to design for a constant per-cell retention time (64 ms) and refresh interval (t_{REFI}), changing instead the time that each refresh command takes to complete (t_{RFC}). Essentially, a strategic decision was made to refresh a larger number of rows per refresh command, rather than issuing more refresh commands.

As t_{RFC} increases, concern has been raised about the performance impact of refresh [44, 64]. When a refresh command is issued, it blocks a full rank, decreasing available parallelism in the memory subsystem significantly. Read requests to the rank currently being refreshed stall until the refresh command completes, affecting system performance. To attack this problem, recent approaches rely on scheduling refresh commands at opportune times [64], or issuing refresh commands less frequently than specified by the DRAM manufacturer [44]. (We comment on related work in Section 3.3.)

3.1.1 DDR-4 DRAM and Fine Granularity Refresh

First discussed in US Patent 2012/0151131 [36], a Fine Granularity Refresh (FGR) feature has been recently announced as part of the JEDEC's DDR-4 DRAM specification [33]. FGR attempts to tackle the increase in t_{RFC} by creating a range of refresh options for memory controller use: {1x, 2x, 4x}. 1x refresh is a direct extension of DDR-2 and DDR-3 refresh: each refresh command takes t_{RFC}, and it must be issued every t_{REFI}=7.8 μs. 2x and 4x modes require that refresh commands be issued twice and four times as frequently–at 3.9 and 1.95μs intervals, respectively. However, in these modes, fewer DRAM rows are refreshed per command, and as a result, their refresh cycle times t_{RFC2x} and t_{RFC4x} are shorter (although not proportionally). Table 3.1 lists the programmable refresh intervals specified for DDR-4 DRAM.

3.1.2 Contributions

This chapter makes the following contributions:

– We conduct an analysis of DDR-4 DRAM's FGR, and determine that there is no single mode that works well across all the applications studied.

– We propose *Adaptive Refresh*, a simple yet highly effective mechanism that leverages DDR-4 DRAM's FGR, by dynamically choosing the mode that suits best each application, and each phase within the application.

– We introduce *Command Queue Seizure*, a phenomenon that we show will become a concern in systems built with high-density DRAM chips (16 and 32 Gb),

which are expected to be common for DDR-4 DRAM and beyond. Command Queue Seizure occurs when the memory controller issues a refresh command to a rank, only to find that it is soon unable to process memory requests to other ranks because the command queue is taken up by commands associated with the rank being refreshed (and thus ineligible for processing).

– We propose *Delayed Command Expansion* and *Preemptive Command Drain*, two complementary mechanisms to address the Command Queue Seizure phenomenon. In Delayed Command Expansion, the memory controller withholds expansion of memory requests into the command queue if such requests are to ranks that will be refreshed soon. In Preemptive Command Drain, the scheduler prioritizes commands in the command queue that map to ranks about to be refreshed, so that they may leave the queue and not occupy valuable slots during the upcoming refresh. The net result is a higher availability of the command queue to memory requests that can proceed concurrently to specific refresh actions.

Our results show that AR does exploit DDR-4's FGR effectively. However, once our proposed DCE and PCD mechanisms are added, DDR-4's FGR becomes redundant in most cases, except in a few highly memory-sensitive applications, where the use of AR does provide some additional benefit. In all, the proposed mechanisms yield 8 and 14% performance gains with respect to traditional refresh at normal and extended DRAM operating temperatures, respectively.

Table 3.1: Refresh cycle times (t_{RFC} = amount of time each refresh command takes) and refresh intervals (t_{REFI} = how frequently refresh commands must be issued) in the DDR-4 DRAM specification [33]. Values for large chips are extrapolated.

Chip Size	t_{RFC_1x}	t_{RFC_2x}	t_{RFC_4x}
8 Gbit	350 ns	260 ns	160 ns
16 Gbit	480 ns	350 ns	260 ns
32 Gbit	640 ns	480 ns	350 ns
t_{REFI}	7.8 μs	3.9 μs	1.95 μs

3.2 Background

Modern DRAM-based memory systems are organized with many degrees of parallelism. Each microprocessor chip has one or more memory controllers, which control one or more *channels*. Each channel has a small number of *ranks*, where each rank contains a handful of DRAM chips (including ECC). Within each chip are a handful of *banks*. All chips in a DRAM rank respond in tandem to commands, meaning a memory controller has scheduling parallelism opportunities across banks, ranks, and channels.

For server memory systems, total system capacity is a primary design objective. Logically speaking, a memory controller may be able to support as many as 32 ranks (2 LRDIMMs with 2 physical ranks/DIMM and 8-high DRAM stacks per rank). However, electrically, channels are much more constrained, due to I/O channel signaling limitations. The faster a channel is run, the more power is burned in signaling and termination. In this chapter we simulate an energy-efficient server-class configuration: a four-rank system at a power-friendly data rate of 1,600 Mbps. In the future, lower-power, higher-frequency offerings may

be possible through 3D TSV-Stacked Master-Slave (3DS) technology [9].

3.2.1 JEDEC DDR4 DRAM Specification

The initial JEDEC DDR4 DRAM specification was released in September 2012 [33], and initial server products with DDR4 DRAM chips are anticipated to begin shipping in 2014 [74]. There are several key new features of this standard: memory speeds are expanded to 3,200 Mbps (compared to 1,600 Mbps for the initial DDR3 specification) with high-tap (V_{DDQ}) termination; core and I/O supply voltage (V_{DD}/V_{DDQ}) are lowered to 1.2 V (compared to 1.5 V for DDR3 and 1.35 V for DDR3L) with a new supply (V_{PP}) added for word-line voltage; several reliability features are added, such as write data CRC, command/address bus parity, and dynamic bus inversion; and architecturally, a key change is the concept of a *bank group*, with the number of banks increased from eight (DDR3) to sixteen. Two (x16) to four (x4/x8) bank groups per chip allow interleaving of column access operations between different bank groups at the periphery blocks (data lines and data sense amplifier) keeping column cycle time (t_{CCD}) to the minimum timing (4 cycles). To help with core power, the page size for x4 parts will be cut in half, dropping from 1 kByte to 512 Bytes. Several standby power reduction features are also added, such as gear down mode, chip-select-to-address latency, and maximum power saving mode.

3.2.2 DDR4 DRAM Refresh Challenges

As technology scaling advances, DRAM tends to maintain the same number of electrons in its storage capacitor (Yoon and Tressler [75] estimate on the order of 100,000), so unlike NAND technology, which will soon reach tens of electrons per floating-gate cell [75], DRAM can theoretically continue to scale. However, concerns about the end of scaling remain well founded, since it will become increasingly difficult to maintain the same amount of charge in the storage capacitor in future technologies. One major reason is that the per-cell capacitance will decrease as lithographic scaling results in physically smaller DRAM cells, but the aspect ratio of the deep-trench cell capacitor cannot keep growing. Supply voltage scaling to meet power reduction demands also causes charge loss. Increased series resistance of the cell access transistor and the bit line (due to smaller cell geometry) makes restoration of charge into the cell slower, and leaves insufficient charge in the storage capacitor. In addition to a decrease in charge, the input offset voltage of the sense amplifier also increases due to increased variability in its transistor pair, which makes it more difficult to sense data, even if the amount of charge is the same.

All of these scaling issues make it more difficult to meet JEDEC's DRAM cell retention time specification of 64 ms in future technology. Thus, in addition to increases in t_{RFC} as DRAM chip density increases, it is anticipated that t_{REFI} may also worsen.

To tackle the refresh problem, DDR4 includes several new concepts, one of which we leverage in this chapter: Fine Granularity Refresh. In addition, DDR4 includes low power auto self-refresh and temperature-controlled refresh mode, but these are related to saving refresh power at low temperature and when

idling, and are not within the scope of this work.

3.3 Related Work

RAIDR – Liu et al. propose RAIDR [44], a clever attempt to minimize refresh operations by exploiting inter-cell variation in retention time. The authors assert that only a small number of weak cells require a conservative refresh interval of 64 ms. DRAM rows are grouped into several retention bins, based on the measured minimum retention time across all cells in the corresponding row. Rows are then refreshed at different rates based on the bin they are classified in.

While this may be well suited to certain domains, we believe there are four important reasons why caution is in order when operating DRAMs outside the standards specified mechanisms for server systems, where data integrity is frequently a non-negotiable design constraint.

– Corner sensitivity: The manifestation of a retention failure is dependent not just on the amount of charge retained in a DRAM cell capacitor, but also on the ability of the sensing mechanism to distinguish a 0 from a 1. Retention characteristics are therefore highly dependent on the combination of temperature, voltage, and DRAM internal noise encountered during a particular test, because the effects which must be considered are not just at the DRAM cell, but also in the sensing mechanism. While chip temperature is something which can be measured during an in-system chip characterization run, a system operator generally has little control over the voltages and noise in the memory subsystem, and no ability to know the internal noise sensitivities of a part (since these

will be highly design-dependent). The weak-cell conclusions of a particular characterization pass may thus not represent worst-case retention characteristics, and it is very challenging for a system-level test to control voltage supply variability and internal DRAM noise.

– Manufacturer freedom: JEDEC specifications only dictate a command called "Refresh" along with its duration, necessary frequency, and how to measure the current consumed during this operation (I_{dd5}). Manufacturers are actually free to do whatever needs to be done to maintain the DRAM during the t_{RFC} duration of the refresh command. From this perspective, an assumption that a refresh command may only refresh some rows, or may only refresh a particular number of rows during t_{RFC}, is technically unsafe. While these assumptions may be accurate for certain DRAMs, they are not guaranteed to apply over all DRAMs across manufacturers, and across the lifetime of any standard.

– Access-pattern-dependent mechanisms: A DRAM cell's or row's retention time of its contents may not only be related to the last time at which it was written, but may also be sensitive to the data pattern which is stored or the access patterns to surrounding cells.

– VRT: Variable Retention Time is not a single phenomenon, but rather a general characterization of DRAMs, which states that there are factors which cause some percentage of DRAM cells to exhibit different retention characteristics at different points in time (e.g., exposure to very high temperature during component assembly). VRT effects occur both during manufacturing and in the field. While ECC and other error-tolerance mechanisms are designed to handle intermittent errors, extending the interval at which cells are refreshed introduces the

risk that total error accumulation, due to VRT, standard retention, soft errors, and other effects, may rise above the initial design threshold for a system.

The combination of these four factors means that it may be risky to operate DRAM cells at a refresh interval beyond the specification range that DRAM manufacturers guarantee using their own test methodology. We believe it is crucial to address the refresh challenge through alternate means, for those systems where data integrity is essential. For systems where the risk of weak-cell migration or incorrect characterization is acceptable, the work presented in this chapter is largely complementary to prior refresh interval extension approaches, and can be used in conjunction with those techniques.

Elastic Refresh – JEDEC specifications allow some flexibility in issuing refresh commands: up to eight refresh commands can be postponed or issued in advance. Stuecheli et al. [64] propose Elastic Refresh, a technique that effectively exploits the flexible dynamic range allowed in the JEDEC refresh specification, by being less aggressive in issuing refresh commands. The primary idea behind elastic refresh is to use predictive mechanisms that decrease the probability of a read or a write operation from colliding with a refresh operation. Earlier techniques make use of the flexibility in issuing refresh operations by scheduling them any time the bus or the rank queues are idle. The elastic refresh algorithm extends this concept by waiting an additional period after the rank becomes idle before it issues the refresh operation. This additional idle delay not only reduces the priority of the refresh command, but also exploits bursty behavior in applications.

Although Elastic Refresh lowers the priority of refresh commands and tries

to reduce collisions with reads and writes, as DRAM scales, the increase in t_{RFC} effectively reduces the probability of avoiding such collisions. In this sense, our Adaptive Refresh mechanism is better suited for adapting to bursty behavior in systems and avoiding collisions by moving between DDR4 DRAM's 1x and 4x modes. Additionally, the concept of Elastic Refresh can be easily integrated with our proposed Adaptive Refresh scheme.

Smart Refresh and Selective DRAM Refresh – Ghosh et al. [25] and Song [63] propose similar techniques that eliminate unnecessary DRAM refresh commands and overheads. The Smart Refresh algorithm proposed in [25] maintains a "refresh-needed" counter for each row that gets reset every time the row gets read out or written to. Smart Refresh relies heavily on data access patterns of the workloads, as the number of refreshes issued only reduces if a large number of rows are being activated. Additionally, it has a high area overhead: for a 2 GB DRAM module, it requires 131,072 counters, each 3-bit wide. Song's Selective DRAM Refresh [63] proposes the use of a reference bit for each DRAM row, which is set when being accessed. During refresh, if a row's reference bit is set, refresh is skipped. This proposal suffers from similar issues as Smart Refresh.

RAPID and Flikker – RAPID, proposed by Venkatesan et al. [68], is a software solution that exploits retention time variation among different DRAM cells. The primary idea is to allocate pages with longer retention time before those with shorter retention time. This allows selection of a refresh period that is dependent on the shortest allocated page retention. RAPID risks similar problems as RAIDR, caused by dynamic variation in retention time and DRAM scaling. In addition, its performance is based on the utilization of the memory pages. Flikker [45], another software solution, proposes partitioning data into critical

and non-critical groups. Non-critical data is then refreshed at much lower rates than critical data. However, identifying data criticality requires substantial programmer effort and is orthogonal to our Adaptive Refresh mechanism.

3.4 Understanding DDR4 DRAM's Fine Granularity Refresh (FGR)

The internal operation of DRAM during refresh includes activating some number of pages, waiting for data to be fully restored, and then precharging those pages. The number of pages that are refreshed together depends on the device density, and can be calculated as (Density/PageSize) $\times$ (t_{REFI}/64 ms). This corresponds to 256 pages on a 16 Gb x8 DDR4 DRAM chip for the 1x mode.

Activating a large number of pages simultaneously generates more sensing current than the internal regulator or the power delivery network can sustain. Therefore, refresh pages are internally divided into subgroups, and refresh is performed per subgroup sequentially, with some time delay between subgroups to reduce the peak noise current. For example, 256 pages can be divided into 32 subgroups, with each group having 8 pages, and the 32 subrefresh operations are staggered every 10 ns (manufacturer and technology dependent).

For a normal activation-precharge operation, the minimum row cycle time is denoted as t_{RC} cycles, and determines the minimum time between accesses to different rows; but for refresh operations, the cycle time is longer than t_{RC}, as each sub-refresh operation activates more than one page and generates more sensing noise. We denote this as $t_{\mathrm{RC\_Refresh}}$. When the subrefresh staggering

delay t_{STAG} is added, the time to complete one refresh operation in 1x mode can be expressed as $t_{RFC\_1x} = (N-1) \times t_{STAG} + t_{RC\_Refresh}$, where N represents the number of subgroups.

For the 2x or 4x modes, there are N/2 or N/4 subgroups, respectively. Thus, the time to complete one refresh operations in 2x mode is $t_{RFC\_2x} = ((N/2) - 1) \times t_{STAG} + t_{RC\_Refresh}$, and similarly for 4x mode.

Note that t_{RFC_2x} is longer than half of t_{RFC_1x}, and that t_{RFC_4x} is also longer than half of t_{RFC_2x}, both due to the term $t_{RC\_Refresh}$. In other words, for each subrefresh operation, there is a startup and completion overhead time $t_{RC\_Refresh}$ which must be amortized. For the 2x and 4x modes, this is amortized over a smaller number of refreshes, so the total time spent doing refreshes in 2x and 4x mode grows, as this initiation cost is paid 2x and 4x more frequently. This introduces a tradeoff: total DRAM stall time due to refresh is minimized when long-latency (many-row) 1x mode is used, but the average stall time should be smallest when shorter refresh operations are used, allowing arriving reads to be issued as soon as possible.

3.4.1 FGR Characterization

To understand the impact of FGR on performance, we run experiments on a set of parallel applications using the 1x, 2x and 4x refresh modes. (A detailed description of the experimental methodology can be found in Section 3.7.) Figure 3.1 shows the corresponding performance data. From the plot we see that, on average, the 1x mode tends to perform better than both the 2x and 4x modes. However, for certain applications like *art* and *swim*, the 4x mode tends to per-

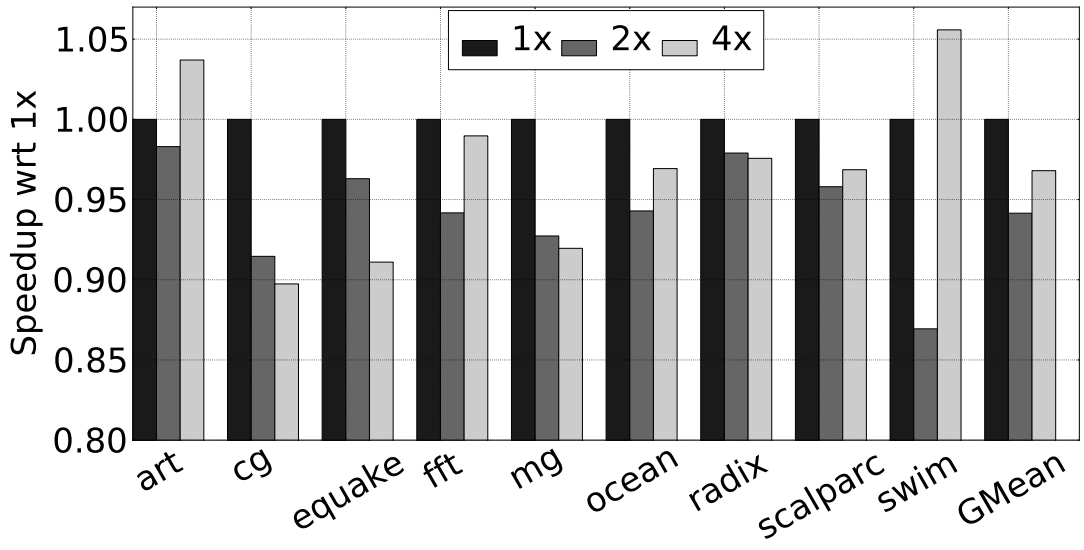

Figure 3.1: Performance (higher is better) of the 1x, 2x and 4x modes, running in the normal temperature range (below 85°C), normalized to the 1x mode.

form better than both 1x and 2x modes. We also observe that 2x is always either the second best or the worst performing among all modes. The reason is because, when moving from 1x to 2x to 4x, while t_{REFI} scales linearly, t_{RFC} doesn't, and this makes the 2x mode sub-optimal. Therefore, for the remaining part of the chapter we do not consider the 2x mode for our experiments

In order to gain insight as to why certain applications performed better in the 1x mode and others performed better in the 4x mode, we choose two applications to analyze: *swim* (which has better 4x performance) and *equake* (which has better 1x performance). We find that for memory-intensive applications, there is a clear benefit from short-latency refresh operations (4x mode). This is because the command queue will often receive requests for a rank that is being refreshed, and these commands will sit in the queue waiting for refresh to complete. The result is a longer average DRAM access time. Figure 3.2 shows the

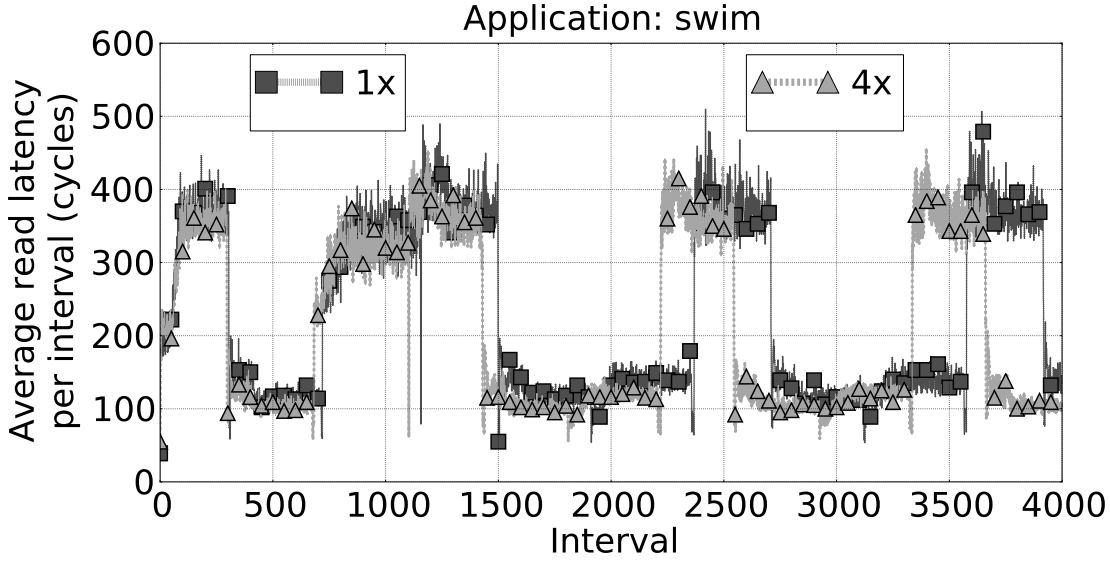

Figure 3.2: Average read latency for *swim* in 1x and 4x FGR modes (lower is better for performance).

average read latency for *swim* (from the SPEC-OMP parallel suite) when running in 1x and 4x modes. 4x mode has on average a smaller read latency than the 1x mode, because DRAM commands spend less time waiting for a refresh to complete in the 4x mode, therefore improving performance.

On the other hand, for applications with low memory utilization, longer refresh commands are less disruptive to overall performance. The cumulative time spent doing refreshes is lower for long-latency commands–more rows are refreshed per command, thus better amortizing $t_{\text{RC\_Refresh}}$. Figure 3.3 shows the average number of cycles the memory controller remains idle due to refresh for *equake* (also from the SPEC-OMP parallel suite) when running the 1x and 4x refresh mode configurations. The 4x mode yields more idle cycles than the 1x mode, which affects performance negatively.

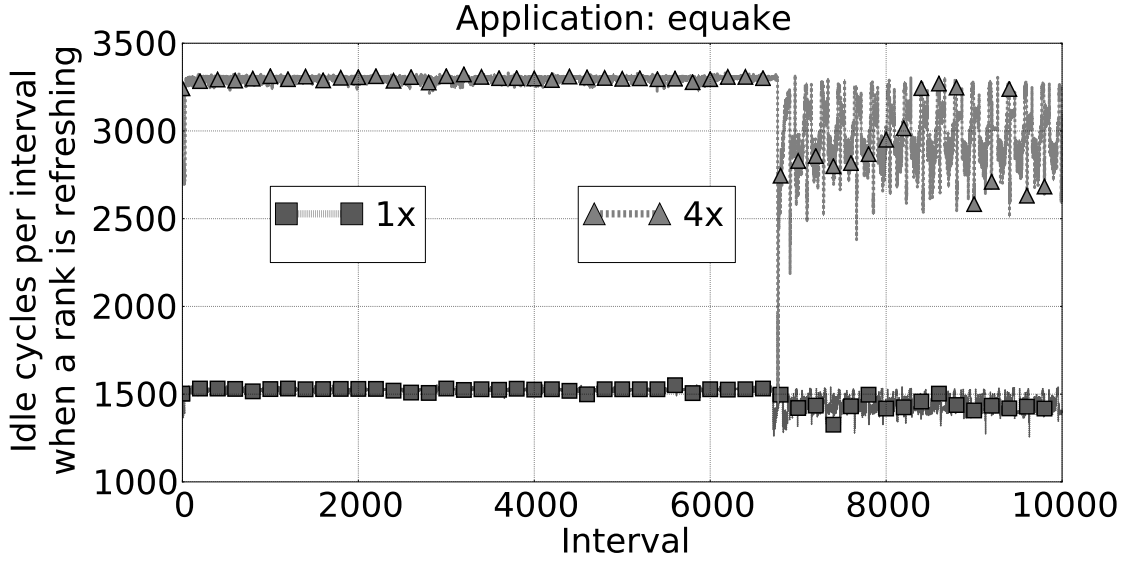

Figure 3.3: Number of cycles the controller remains idle while a rank is refreshing (lower is better for performance) for *equake* in 1x and 4x FGR modes.

3.5 Adaptive Refresh

From our above analysis, we conclude that no single refresh mode works best in all cases. Thus, we set out to design an adaptive FGR mechanism capable of determining the best refresh mode for a particular workload during the course of its execution.

We propose a very simple Adaptive Refresh mechanism to achieve this goal. The mechanism tracks data bus utilization as a proxy measurement for performance, because it is directly observable at the memory controller, and because it tends to correlate strongly with system performance for memory-bound applications. Specifically, at each interval, we use a counter that is incremented every time the memory controller issues a read or a write—the two commands that involve an actual data transfer.

Algorithm 1: Simple FGR-Aware Adaptive Refresh

```
1: while application is still running do
2:     Choose 1x mode and run the application for n intervals, while monitoring data
       bus utilization
3:     Choose 4x mode and run the application for n intervals, while monitoring data
       bus utilization
4:     if utilization in 1x mode ≥ utilization in 4x mode then
5:         Choose 1x mode and run the application for m ≫ n intervals
6:     else
7:         Choose 4x mode and run the application for m ≫ n intervals
8:     end if
9: end while
```

Algorithm 1 shows the simple procedure for our Adaptive Refresh mechanism. We start by initially running the application in the 1x mode for a training period of n intervals, while monitoring data bus bandwidth. At the end of n intervals, we switch from the 1x mode to the 4x mode, and train for a period of another n intervals, again while monitoring data bus bandwidth. At the end of these $2n$ intervals, we compare the measured utilization for both refresh modes, and pick the mode that has yielded the higher utilization. We then continue to run using the chosen mode for a period of $m \gg n$ intervals. This whole process is repeated periodically in order to accommodate changes in phase behavior of the application.

Picking the right n, m, and interval duration is important. In our experiments, we empirically determined the refresh rate for the 1x mode (t_{REFI}) to be a good value for our algorithm's interval. This is convenient because a refresh

command is sent out every t_{REFI} cycles in the 1x mode (twice/four times as often for the 2x/4x modes), so picking this interval trivially guarantees that the controller does not miss any refreshes, even when switching modes. As for n and m, we empirically determine that $n = 5$ and $m = 100$ is a good compromise when all the associated overheads are accounted for. We evaluate this Adaptive Refresh mechanism in Section 3.8.1.

Micro-architectural Support for Adaptive Refresh

Very few minor additions are required to a memory scheduler in order to incorporate Adaptive Refresh. Note that a scheduler already has registers that store the current values of t_{RFC} and t_{REFI}. Two additional registers are required to store the value of the training and testing intervals: n (3-bit register) and m (7-bit register). Another 7-bit register and a 7-bit adder are needed to keep track of the elapsed intervals during the training and testing phases. Modern-day memory controllers already have the capability of measuring data bus utilization, and therefore no additional hardware is required to monitor the bandwidth of the data bus. However, a 13-bit adder and two 15-bit registers are required to keep track of the cumulative utilization across n intervals during the two training phases. Finally, a 15-bit comparator is necessary to compare the utilization measured during training in the 1x and 4x modes and a 2-to-1 multiplexor is needed to choose the FGR mode based on the output of the comparator.

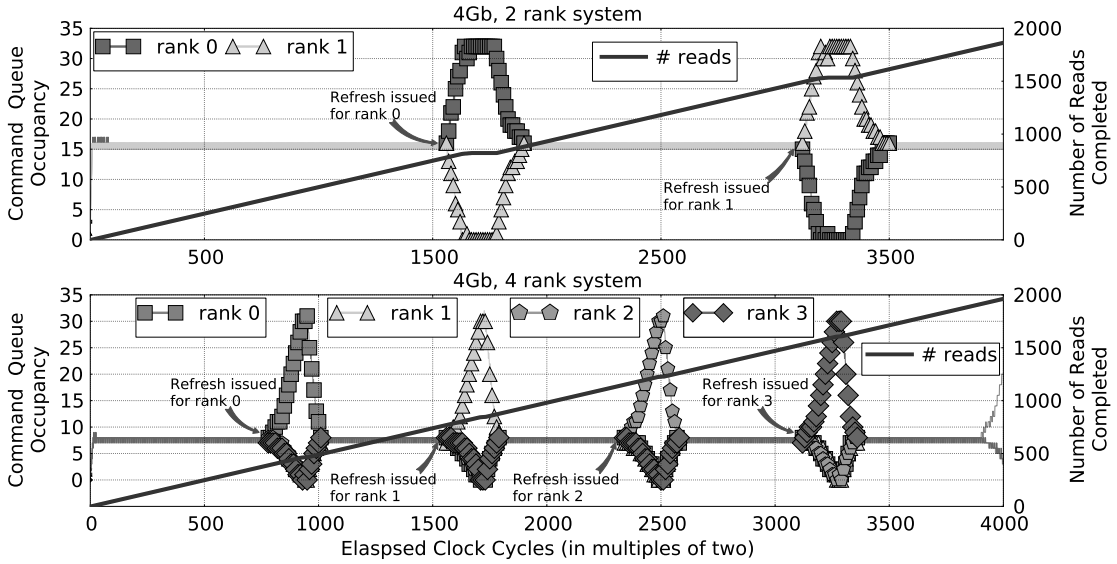

Figure 3.4: Analysis of the command queue seizure phenomenon for a 4 Gb DRAM chip running a micro-benchmark with an even distribution of loads and stores across ranks and banks. For a two-rank system, the command queue can fill up with DRAM commands to a rank being refreshed, momentarily stalling command issue and increasing the idle time of the scheduler, thereby hurting performance. For a four-rank system, the problem is alleviated with sufficient command variety in the queue, and the controller is able to continue to issue commands while a refresh operation completes in a target rank.

3.6 Increasing The Command Queue Effectiveness of High - Density DRAM

During a typical refresh operation, the controller cannot issue any DRAM commands to a rank while it is being refreshed. Fortunately, modern server memory systems have multiple ranks, and thus when a refresh operation to a rank is underway, the controller can still issue DRAM commands to the other ranks that are not being refreshed. In earlier technology generations like DDR3 DRAM, the refresh latency t_{RFC} was still small enough that issuing DRAM commands

to non-refreshing ranks could likely keep the controller busy. However, for very dense DRAM chips, this may no longer be the case.

Recall that, as DRAM's chip density increases, the time required to refresh these DRAM chips also increases. As refresh latency begins to grow, we notice that our modeled controller remains idle for a longer period of time despite the presence of multiple ranks in the memory system. In this section, we introduce and analyze a new phenomenon caused by refresh commands on the memory controller: *command queue seizure*.

For a memory system designed with multiple ranks, refreshes are staggered across the ranks. This is done primarily for two reasons: (1) Issuing simultaneous refreshes to all ranks can push memory systems close to (or over) their peak power budgets (or cause voltage drop issues). This is because refresh is the most power-hungry DRAM operation, and the V_{dd} voltage rail is commonly shared across all chips in the system. Although not specifically reported in DDR JEDEC specifications, almost all the modern multi-rank memory controllers of which we are aware stagger refreshes among ranks for this reason (because it is a system design issue involving power budgets). (2) From a performance perspective, staggering refreshes to multiple ranks ensures that the controller does not remain idle while a rank is being refreshed. Due to the complexity involved in the design of command queues for on-chip memory controllers in current memory systems, a majority of system designs opt for a common command queue for all requests targeting any DRAM rank on a given memory channel [21].[1] For a high-performance memory system, addresses will be hashed across channels and ranks, enabling maximum system parallelism. An understanding of the im-

[1] Additionally, experiments conducted using per-rank command queues (as opposed to per-channel command queues) did not provide a significant improvement in performance to warrant the need.

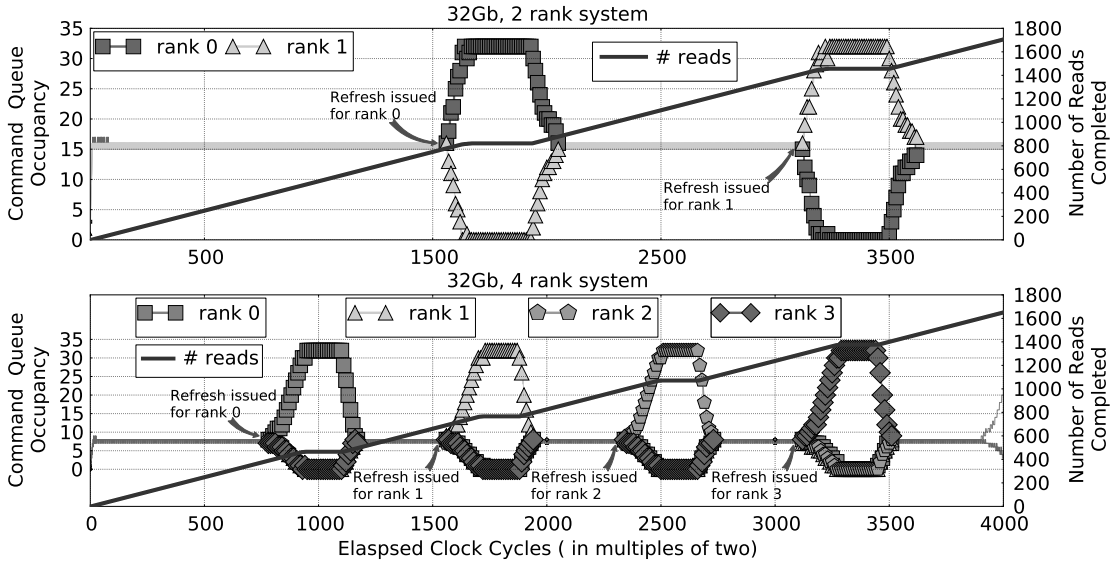

Figure 3.5: Analysis of the command queue seizure phenomenon for a 32 Gb DRAM chip running a micro-benchmark with an even distribution of loads and stores across ranks and banks. In large capacity DRAM chips, for a two-rank system, once the command queue has been filled with commands to the rank being refreshed, long $t_{\mathbf{RFC}}$ times quickly leads to the memory scheduler stalling for longer. The problem worsens for four-rank systems as the number of refresh commands issued per refresh interval doubles. The net result is an increase in the idle time of the scheduler, which leads to a loss in performance.

pact of refresh on the command queue can be obtained by looking at Figures 3.4 and 3.5. These plots show the effect of command queue seizure when running a micro-benchmark with an even distribution of commands arriving to each rank, for a system modeled using 4 Gb and 32 Gb DDR4 DRAM chips.

In Figure 3.4, we see the command queue occupancy and the number of reads being issued per cycle for a two-rank and four-rank system using a 4 Gb chip. Let us first analyze the two-rank system. In interval 3120, a refresh is issued to rank 0. It remains idle for the duration of refresh while the scheduler issues commands to the other rank in the system, rank 1. However, due to

the even distribution of the memory stream, the controller will steadily drain the command queue of rank 1 requests, while filling it with requests to rank 0. When the command queue gets filled up with requests to rank 0, the controller stalls, as it can no longer issue any more commands. This is indicated by the plateaus in the "# reads" line, which tracks the cumulative count of issued read operations to all locations in the memory system. For a four-rank system using 4 Gb chips, this poses less of a problem, as the extra ranks add sufficient variety of commands to the queue's available scheduling resources. The queue seldom fills up with pending commands to the rank being refreshed, and the controller is continuously able to issue operations to the memory system ("# reads" increases monotonically).

The situation changes significantly if we consider the increased refresh cycle times of high density DRAMs (32 Gb chips) (figure 3.5). Here the increased refresh latency (t_{RFC}) blocks the rank being refreshed for a longer period of time. This gives the command queue more time to fill up with commands to the rank being refreshed. The situation becomes untenable for high-density memory with longer refresh latencies, and causes the controller to stall for relatively long periods of time. The phenomenon actually worsens for a four-rank system when compared to a two-rank system, as the number of refresh commands sent in one refresh interval (t_{REFI}) is also twice as many.

3.6.1 Preemptive Command Drain

In this section we propose a new scheduling technique that helps mitigate the negative effects of command queue seizure, which we call *Preemptive Command*

Drain (PCD). The idea behind PCD is to drain the command queue of commands to a rank that is about to be refreshed, by prioritizing them over commands to other ranks. This is easily accomplished by checking whether the refresh countdown of each DRAM rank is below a certain threshold. In this way, when the refresh operation is actually issued by the controller, there will be fewer (possibly none) commands to the refreshing rank in the command queue, and more commands to non-refreshing ranks. This gives the memory controller more opportunities for successful command scheduling while the refresh operation is taking place, thereby reducing idle cycles.

For this to be effective, it is important to pick the right refresh countdown threshold. In the general case, we could use a simple interval adaptation scheme, similar to the one proposed for Adaptive Refresh (Section 3.5), to determine the right threshold for each application and each application phase. In our experimental setup, however, we conducted a sensitivity study, and found that a threshold of 150-250 cycles was universally optimal across the studied applications and phases. Thus, without losing generality, in our evaluation we show PCD results with a constant threshold of 200 cycles (Section 3.8.2).

Micro-architectural Support for PCD

Most modern memory schedulers use some variant of the FR-FCFS scheduling algorithm proposed by Rixner et al. [59]. The basic FR-FCFS scheduling policy prioritizes CAS commands over RAS commands and to break ties, older over younger commands. In order to incorporate PCD into FR-FCFS, a few minor changes to the scheduling algorithm would be needed. PCD + FR-FCFS would need to prioritizes commands in the following order: (a) CAS commands over

RAS commands for the soon-to-be-refreshed rank; (b) older commands over younger commands for the soon-to-be-refreshed rank; (c) CAS commands over RAS commands for other ranks; and finally (d) older commands over younger commands for other ranks. An 8-bit comparator would be required to check if the refresh countdown of a DRAM rank is below a particular threshold. If so, commands mapping to this rank would then be selected with higher priority.

3.6.2 Delayed Command Expansion

Memory controllers typically hold loads and stores received from the last-level cache in a transaction queue. As space becomes available in the command queue, these memory requests are expanded into the appropriate DRAM commands and transferred to the command queue. The PCD algorithm explained above tackles the negative effects of refresh on the command queue, by prioritizing the scheduling of commands to a rank about to be refreshed, so they can be drained from the command queue instead of getting stuck in place for the duration of the refresh operation. This process can also be optimized on the transaction queue side: We propose that the memory controller be allowed to temporarily put off memory requests to a rank that is being refreshed, instead expanding and transferring to the command queue requests to non-refreshing ranks. This helps increase the number of issuable commands inside the command queue, potentially improving performance. We call this Delayed Command Expansion (DCE), and evaluate it in Section 3.8.2.

Micro-architectural Support for DCE

In our model, when memory requests arrive at the transaction queue, they are expanded into DRAM commands in FIFO order and placed in the command queue. In order to incorporate DCE, an additional comparison of the rank id's of the memory requests with the current rank that is being refreshed would also be needed. As a conservative estimate, this requires a 2-bit comparator for each transaction queue entry. However, it is also possible for multiple entries to share a single comparator, as long as they are used serially. DCE is triggered only when a rank is refreshing. Any memory controller would already have this information stored, and hence additional hardware would not be required to support this. Similar to PCD, the information of whether a rank is being refreshed or not would be determined using a comparator, and if so commands mapping to this rank would then be selected with lower priority for expansion into the command queue.

3.7 Experimental Methodology

3.7.1 Architecture Model

Our baseline processor model integrates eight cores and supports a DDR4-1600 memory subsystem with one independent memory channel (we model our memory subsystem based on the configurations used in IBM Power7^{TM} systems 710, 720 and 730, where the ratio of threads to memory controllers is eight [21]). The DIMM structure and timing parameters of our memory model follow JE-

DEC's DDR4 SDRAM specification [33]. For power and energy calculations, we use I_{dd} values estimated from discussions with authors of the JEDEC DDR4 standard and DRAM chip manufacturers. The micro-architectural features of the baseline processor are shown in Table 3.2; the parameters of the L2 cache, the memory system, and the DRAM power model are shown in Tables 3.3, 3.4 and 3.5.

3.7.2 Simulation Setup and Applications

All our experiments have been carried out by extending the SESC simulation environment [58] with DRAMSim2 [60], which has been modified to model JEDEC's DDR4 specification, including bank and rank groups, and Fine Granularity Refresh. We evaluate a number of configurations as follows: The configuration 1x represents the baseline auto-refresh policy present in current day memory systems; the 4x configuration makes use of the corresponding FGR refresh mode; AR is our proposed adaptive refresh technique that leverages FGR; PCD denotes our proposed Preemptive Command Drain mechanism, wherein the scheduler prioritizes commands in the command queue that are mapped to ranks that are about to be refreshed; and finally, DCE is our proposed Delayed Command Expansion scheme, that withholds expansion of memory requests into the command queue if they are to a refreshing rank. We evaluate our proposed schemes on a set of parallel applications, running eight threads each, to completion. Our parallel workloads constitute a good mix of scalable scientific programs from different benchmark suites, as shown in Table 3.6.

3.7.3 DDR4 Extended Temperature Range

In the normal operating temperature range for DRAMs (below 85°C), the average time between refresh commands (t_{REFI}) is 7.8 μs. DRAMs can also operate in the extended temperature range (between 85°C and 95°C), particularly in server type environments and while using 3DS technology [9, 22]. The required time between refresh commands at high temperatures is halved to 3.9 μs. There is global interest in expanding the operating temperature ranges of data centers, driven mostly by the desire for achieving higher operating efficiency and lower total cost of ownership [10]. Increasing operating temperature ranges in a data center environment causes all components in servers (including memory) to see higher temperatures. Evaluation of the extended temperature range is hence both necessary and critical when evaluating high-density memory systems.

3.8 Evaluation

3.8.1 Adaptive Refresh

Figure 3.6 shows the performance obtained by the 1x, 4x, and AR configurations in the normal DRAM operating temperature range, normalized to the performance of the 1x configuration. Our proposed AR essentially matches the performance of the refresh mode that works best for each individual application. For applications that are not as memory-sensitive (which tend to work better in the 1x configuration), like *mg*, *cg*, *equake* and *radix*, AR outperforms the 4x configuration significantly and is within less than 2% of the 1x configuration.

Table 3.2: Core Parameters.

Technology	32 nm
Frequency	3.2 GHz
Number of cores	8
Fetch/issue/commit width	4/4/4
Int/FP/Ld/St/Br Units	2/2/2/2/2
Int/FP Multipliers	1/1
Int/FP issue queue size	32/32 entries
ROB (reorder buffer) entries	96
Int/FP registers	96 / 96
Ld/St queue entries	24/24
Max. unresolved br.	24
Br. mispred. penalty	9 cycles min.
Br. predictor	Alpha 21264 (tournament)
RAS entries	32
BTB size	512 entries, Direct-mapped
iL1/dL1 size	32 kB
iL1/dL1 block size	32 B/32 B
iL1/dL1 round-trip latency	2/3 cycles (uncontended)
iL1/dL1 ports	1 / 2
iL1/dL1 MSHR entries	16/16
iL1/dL1 associativity	Direct-mapped/4-way
Memory Disambiguation	Perfect
Coherence protocol	MESI
Consistency model	Release consistency

For more memory-sensitive applications like *art*, AR performs better than the 1x configuration, and it is within 1.5% of the performance of the 4x mode. For some applications like *fft*, *ocean*, *scalparc*, and *swim*, AR outperforms both the 1x and the 4x configurations, as AR successfully adapts to application phase changes.

Table 3.3: Parameters of the shared L2 and DRAM.

Shared L2 Cache Subsystem	
Shared L2 Cache	4 MB, 64 B block, 8-way
L2 MSHR entries	64
L2 round-trip latency	32 cycles (uncontended)
Write buffer	64 entries
DDR4 @1600 Mbps DRAM – 16Gb chip size	
Transaction Queue	128 entries
Command Queue	32 entries
Number of Channels	1
DIMM Configuration	Quad rank
Number of Banks	16 per rank
Row Buffer Size	1 KB
Address Mapping	Page Interleaving
Row Policy	Closed Page[a]
Burst Length	8
^{t}**RCD**	10 DRAM cycles
^{t}**CL**	10 DRAM cycles
^{t}**WL**	12 DRAM cycles
^{t}**CCD**	4 DRAM cycles
^{t}**CCD_L**	5 DRAM cycles
^{t}**WTR**	2 DRAM cycles
^{t}**WTR_L**	6 DRAM cycles
^{t}**WR**	15 DRAM cycles
^{t}**RTP**	6 DRAM cycles
^{t}**RP**	10 DRAM cycles
^{t}**RRD**	4 DRAM cycles
^{t}**RTRS**	2 DRAM cycles
^{t}**RAS**	28 DRAM cycles
^{t}**RC**	28 DRAM cycles
^{t}**FAW**	20 DRAM cycles
^{t}**CKE**	4 DRAM cycles

[a]We have conducted our experiments with open page policy and the results and insights closely follow those for closed page, which we present.

Table 3.4: Refresh Parameters

Refresh Parameters: DDR4 @1600 Mbps DRAM – 16Gb chip size	
^{t}**REFI**	7.8 μs
^{t}**REFI-XTemp**	3.9 μs
^{t}**RFC_1x**	384 DRAM cycles
^{t}**RFC_2x**	280 DRAM cycles
^{t}**RFC_4x**	208 DRAM cycles

Table 3.5: Parameters used for 16 Gb DDR4 @ 1600 Mbps DRAM power management features.

IDD0	24 mA
IDD1	32 mA
IDD3P	7.2 mA
IDD2P	6.4 mA
IDD2N	10.1 mA
IDD3N	16.6 mA
IDD4R	60 mA
IDD4W	58 mA
IDD5	102 mA
IDD6	6.7 mA
IDD7	107 mA

Analysis – Let's look into what is happening. We first take an example from the class of applications that are less memory sensitive: *mg*. Figure 3.7 is divided into two plots: The left plot shows the number of cycles per interval the memory controller remains idle while a rank is refreshing (lower is better), in spite of the command queue not being empty, when running the 1x and 4x configurations. The right plot shows the same for AR and the 4x configurations. From the plot on the left, we see that the 1x configuration has fewer scheduler idle cycles than the 4x configuration, which translates into improved system performance. We

Table 3.6: Simulated parallel applications and their input sets.

	Data Mining	[57]
scalparc	Decision Tree	125k pts., 32 attributes
	NAS OpenMP	[12]
mg	Multigrid Solver	Class A
cg	Conjugate Gradient	Class A
	SPEC OpenMP	[11]
swim-omp	Shallow water model	MinneSPEC-Large
equake-omp	Earthquake model	MinneSPEC-Large
art-omp	Self-organizing Map	MinneSPEC-Large
	Splash-2	[73]
ocean	Ocean movements	514×514 ocean
fft	Fast Fourier transform	1M points
radix	Integer radix sort	2M integers

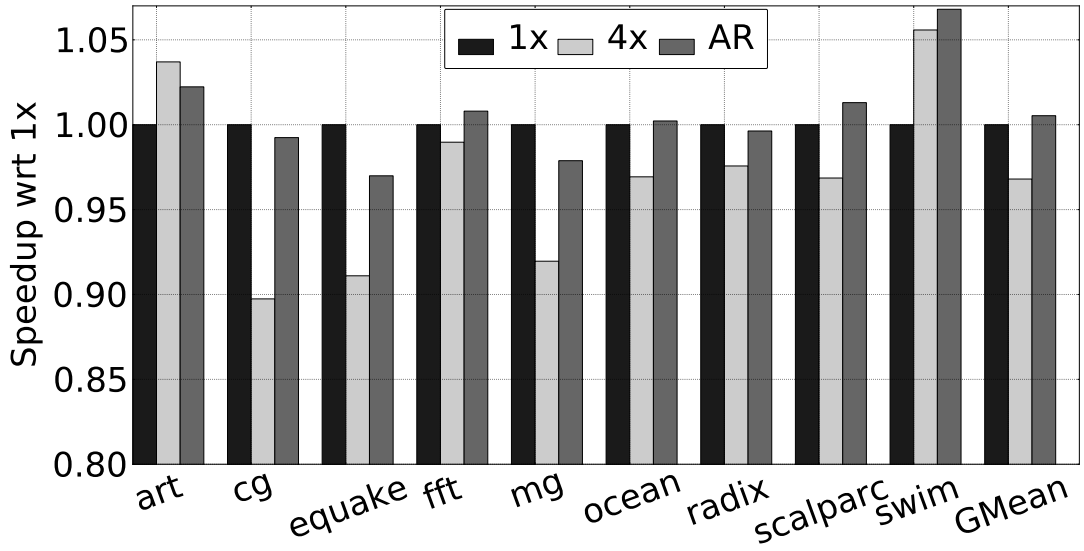

Figure 3.6: Performance (higher is better) in the normal DRAM temperature range for the 1x, 4x and AR configurations, normalized to the performance of the 1x configuration.

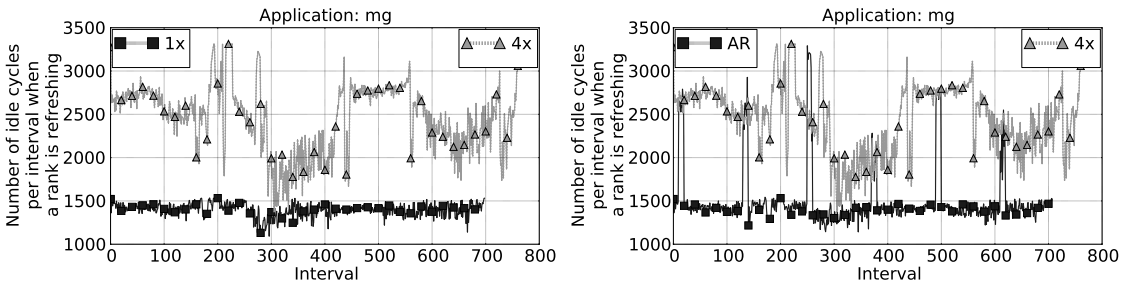

Figure 3.7: Number of cycles per interval the controller remains idle (lower is better) while a rank is refreshing and the command queue is not empty for the application *mg*. The plot to the left shows idle cycles for the 1x and 4x FGR configurations. The plot to the right shows the same for the 4x mode and our proposed AR configuration. AR closely follows the FGR mode that has the fewest scheduler idle cycles (1x in this case), which translates to improved performance.

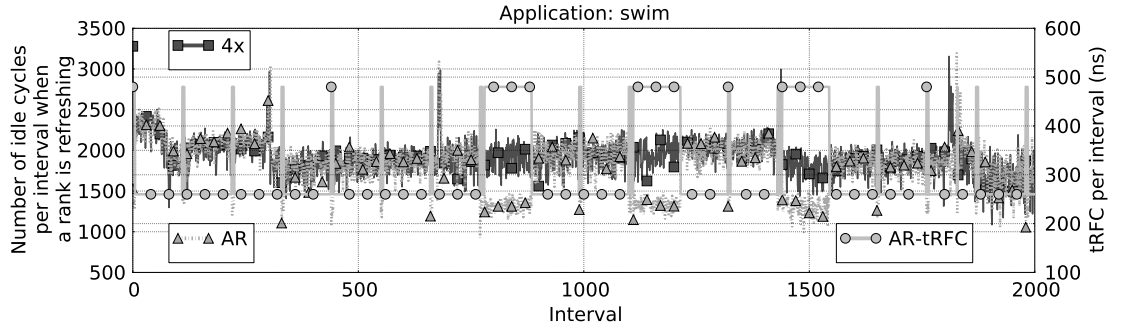

Figure 3.8: Number of cycles per interval the controller remains idle (lower is better) while a rank is refreshing and the command queue is not empty for the application *swim* when running the 4x and AR configurations. The plot shows that AR closely tracks the 4x mode for the most part, but also has periodic drops in idle cycles. These drops correspond to a profitable switch from the 4x to the 1x FGR mode due to a phase change in the application, as indicated by the overlaid t_{RFC} plot which jumps to 480ns from 260ns (when the switch occurs).

can see that AR (right plot) closely tracks the behavior of the 1x configuration. The right plot does show some periodic, narrow spikes for the AR configuration.

These correspond to AR's training phase in 4x mode. As it turns out, *mg* doesn't have phase changes that alter the refresh mode patterns significantly, and 4x mode is never picked by AR. Fortunately, since the training phase is a small fraction of the overall execution, it affects system performance minimally, as shown earlier.

Next we look at an example from the class of applications for which AR performs better than both 1x and 4x modes: *swim*. As before, Figure 3.8 shows the number of cycles per interval the memory controller remains idle while a rank is refreshing (lower is better), when the command queue is not empty. Similar to *mg*, the plot shows that AR closely tracks the better configuration, 4x in this case. However this time AR exhibits wide, periodic drops in idle time. These correspond to adaptation to program phase changes, where AR concludes that a temporary switch to 1x mode is profitable. The overlaid plot of t_{RFC} shows that this is indeed what is happening. The net result is better performance than the static 4x configuration, as shown before (Figure 3.6).

Extended DRAM temperature range – Figure 3.9 shows the performance obtained by the 1x, 4x and AR configurations in extended DRAM operating temperature range, normalized to the performance of the 1x configuration. (Recall that operating in extended DRAM temperature range is an important consideration for high-performance configurations.) In the extended temperature range (XTemp), the volume of refresh commands issued essentially doubles within the same refresh interval. This means that the controller spends more time performing refresh operations. The results show the same trends as in the earlier case of normal DRAM temperature mode, only the performance benefit of picking the right refresh mode is larger, and thus AR is even more beneficial.

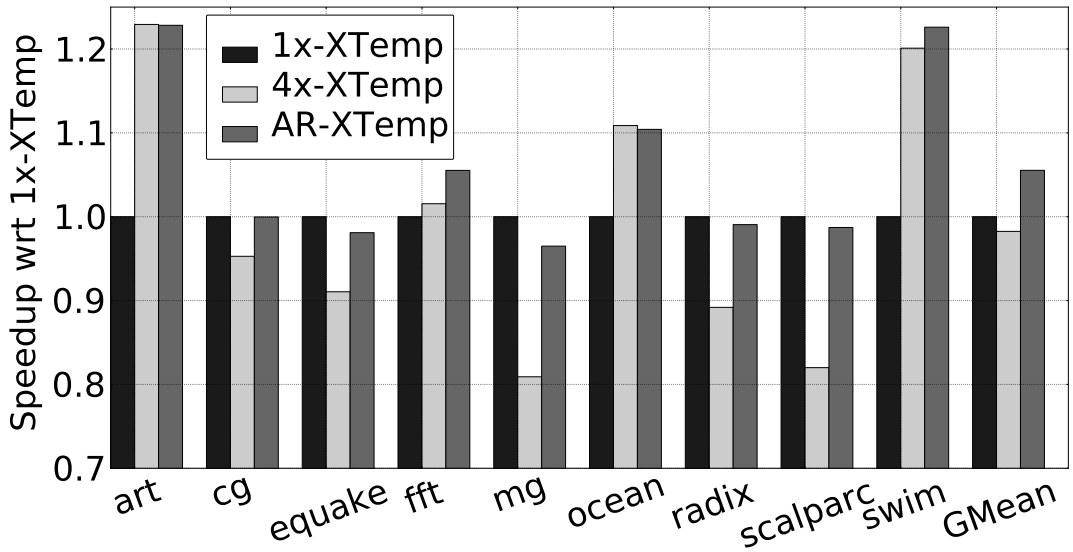

Figure 3.9: Performance (higher is better) in the extended DRAM temperature range for the 1x, 4x and AR configurations, normalized to the performance of the 1x configuration.

3.8.2 Delayed Command Expansion and Preemptive Command Drain

Figure 3.10 shows the performance obtained by the 1x, Delayed Command Expansion in 1x mode (DCE1x), Preemptive Command Drain in 1x mode (PCD1x), and DCE1x+PCD1x configurations, run under normal DRAM operating temperature conditions, and normalized to the performance of the 1x configuration. We see that DCE1x and PCD1x improve performance on average by 3% and 5.5% over the 1x configuration, respectively. Moreover, combining the two schemes has an additive effect: DCE1x+PCD1x shows an improvement in performance of 8% over the 1x configuration.

Analysis – To understand where this improvement is coming from, let's look at *art* a bit more closely. Recall that both the DCE and PCD mechanisms at-

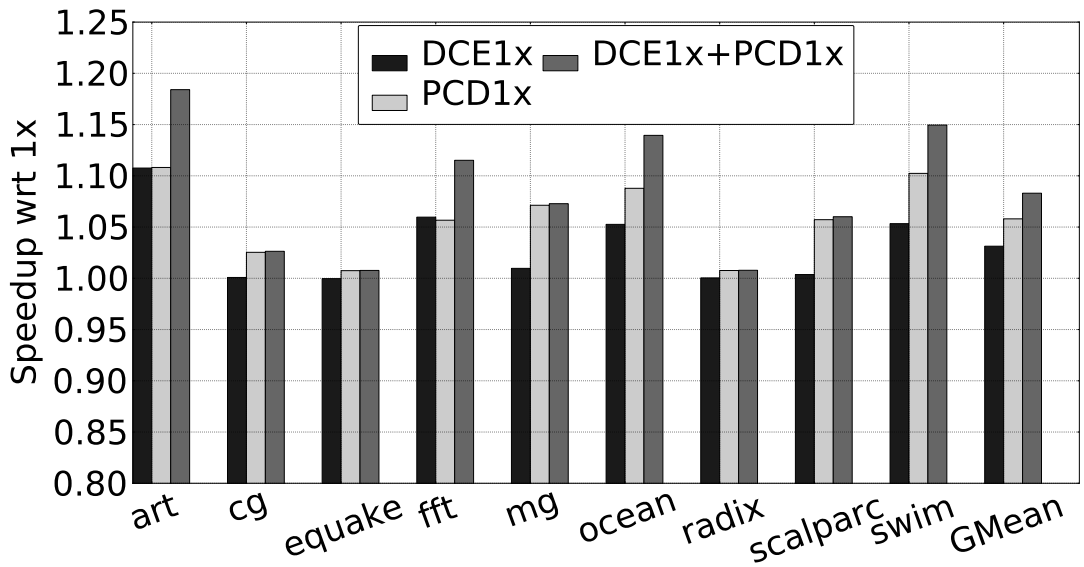

Figure 3.10: Performance (higher is better) in the normal DRAM temperature range for DCE, PCD and DCE+PCD when running in the 1x mode, normalized to the performance of the 1x configuration.

tempt to increase the number of commands to non-refreshing ranks in the command queue, by proactively draining the queue of commands to the refreshing rank, and by prioritizing commands to non-refreshing ranks. Figure 3.11 shows the percentage of command queue slots taken up by commands to non-refreshing ranks per interval for the application *art* when running the 1x and DCE1x+PCD1x configurations. The results show that the configuration DCE1x+PCD1x has a much higher number of commands to non-refreshing ranks in the command queue per interval, when compared to the 1x configuration. This increases the opportunity for issuing more commands per interval, reducing idle cycles in the controller and improving throughput, ultimately improving performance.

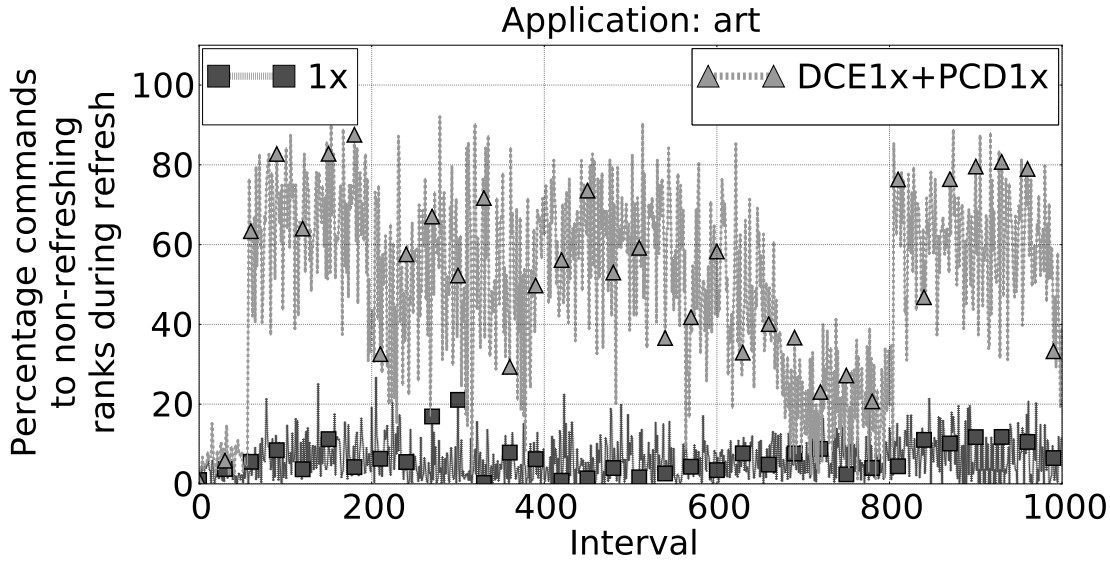

Figure 3.11: Fraction of commands to non-refreshing ranks in the command queue (higher is better) while a rank is being refreshed for the application *art* when running the 1x configuration and the DCE+PCD configuration in the 1x mode.

3.8.3 Putting It All Together: AR+DCE+PCD

Figures 3.12 and 3.13 show the performance obtained by combining our three proposed schemes when running in both normal and extended DRAM temperature ranges, and normalized to the appropriate 1x mode in each case. A few interesting observations can be made from these plots: First, for most applications, adding DCE and PCD to the 4x mode reduces performance when compared to adding the schemes to the 1x mode. This is because the 4x mode cumulatively spends more time on refresh than the 1x mode. We notice that the increase in refresh commands while running the 4x mode increases the idle time during refresh, especially for those applications that do not provide a constant stream of loads and stores (which helps DCE and PCD). Second, following from the above argument, adding AR to DCE and PCD provides no significant

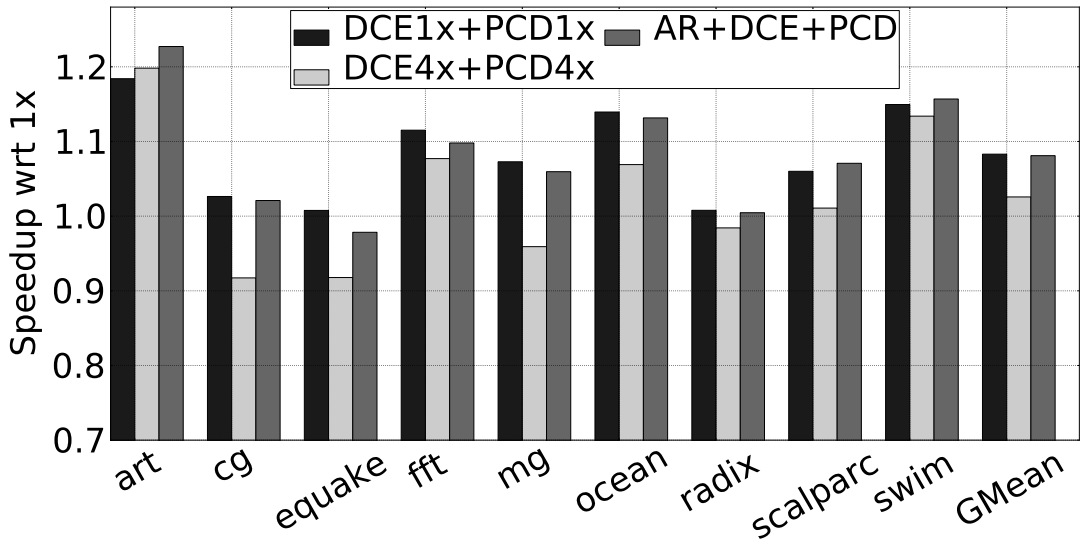

Figure 3.12: Performance (higher is better) in the normal DRAM temperature range when running DCE+PCD in the 1x, 4x and AR modes, normalized to the performance of the 1x configuration.

improvement in performance when compared to adding the schemes to the 1x mode. The combination of AR, DCE, and PCD, improves performance over the 1x configuration by 8 and 14% on average in normal and extended DRAM temperature ranges, respectively. These results are remarkably close (in fact, virtually identical in the case of normal DRAM temperature range) to the ones obtained by DCE1x+PCD1x alone. What this means is that, on average for these applications, leveraging DDR4 DRAM's Fine Granularity Refresh feature offers little advantage once our proposed DCE and PCD mechanisms are in place. For applications that are particularly memory sensitive and/or exhibit refresh phase behavior (*art* and *swim*), we do observe an improvement by adding AR on top of DCE and PCD, especially in the extended DRAM temperature range.

Analysis – To provide further insight, we look at the access pattern of the application *art*, one of the two that benefit from combining AR to DCE+PCD. Fig-

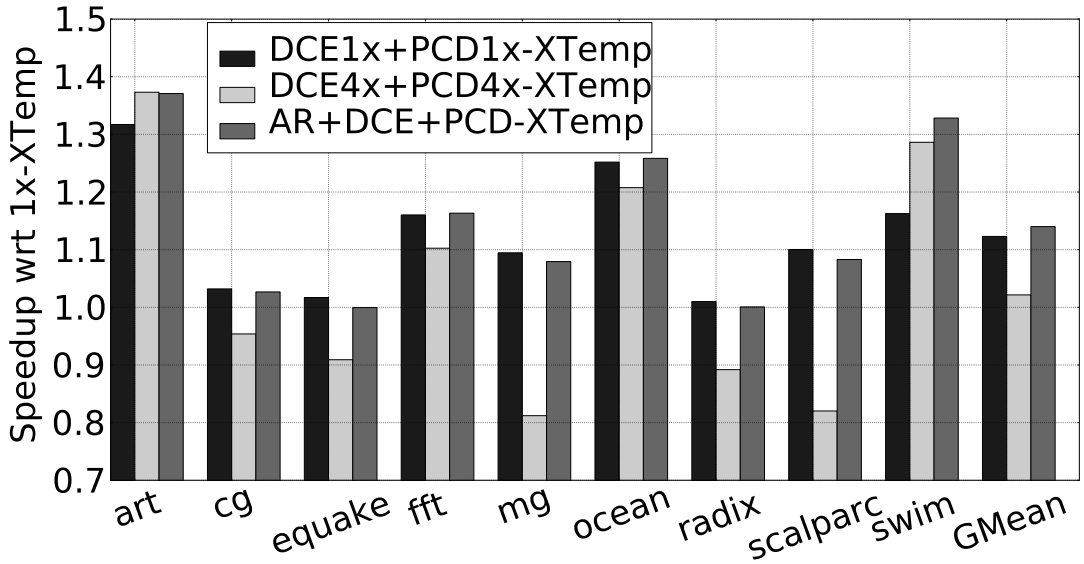

Figure 3.13: Performance (higher is better) in the extended DRAM temperature range when running DCE+PCD in the 1x, 4x and AR modes, normalized to the performance of the 1x configuration.

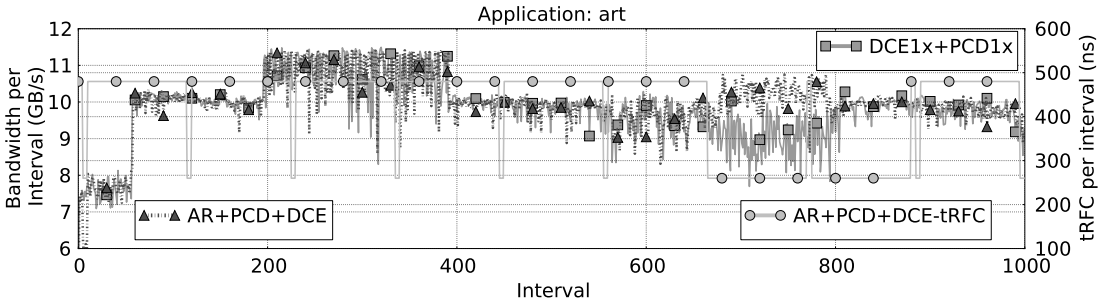

Figure 3.14: Effective data bandwidth for the application *art* when running DCE+PCD in the 1x and AR configurations. AR closely follows the 1x configuration for the most part, but also makes profitable switches to the 4x mode when it finds an opportunity for increasing data bus utilization. This is indicated by the the overlaid t_{RFC} plot in the AR configuration which jumps to 260ns from 480ns when these switches occur.

ure 3.14 shows the effective data bandwidth per interval for the DCE1x+PCD1x and the AR+DCE+PCD configurations on the primary y-axis (left), and t_{RFC}

per interval for the AR+DCE+PCD configuration on the secondary y-axis (right). (For DCE1x+PCD1x, t_{RFC} remains constant at 480 ns.) Around the 650 interval time frame, the application *art* has a memory phase change. AR+DCE+PCD detects this change and immediately switches to the 4x mode. DCE1x + PCD1x, however, remains in the 1x mode. As a result, AR + DCE + PCD is able to sustain a high effective data bandwidth, whereas DCE1x+PCD1x drops in bandwidth at that point.

3.8.4 Energy Calculations

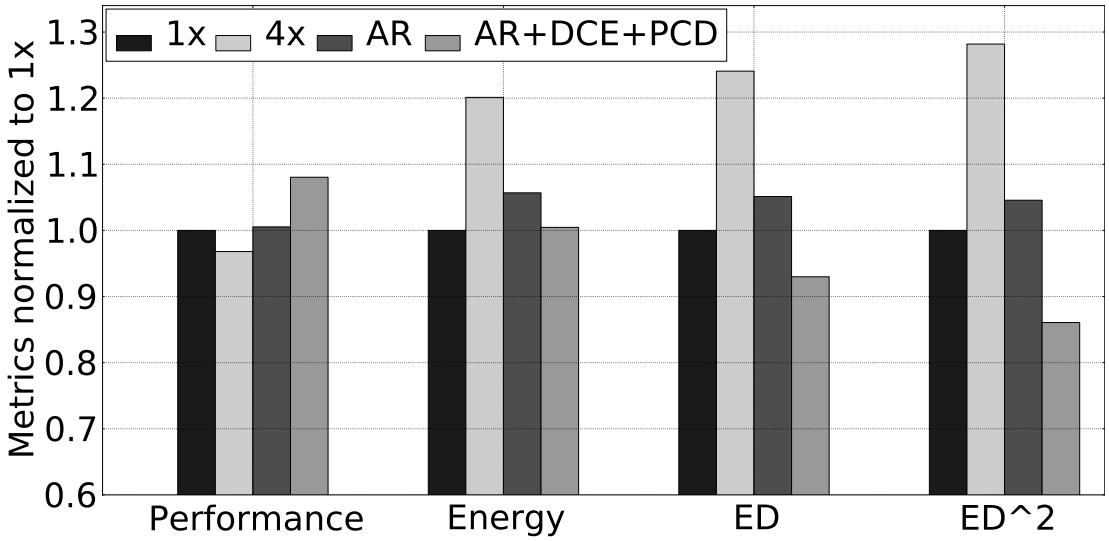

Figure 3.15: Mean performance, energy, energy-delay and energy-delay squared in the normal DRAM temperature range for the 1x, 4x, AR and AR+DCE+PCD configurations, normalized to that of the 1x configuration.

Figures 3.15 and 3.16 show average performance, energy, energy-delay and energy-delay squared numbers for the 1x, 4x, AR, and AR+DCE+PCD configurations, normalized to that of the 1x configuration in each case, in both normal

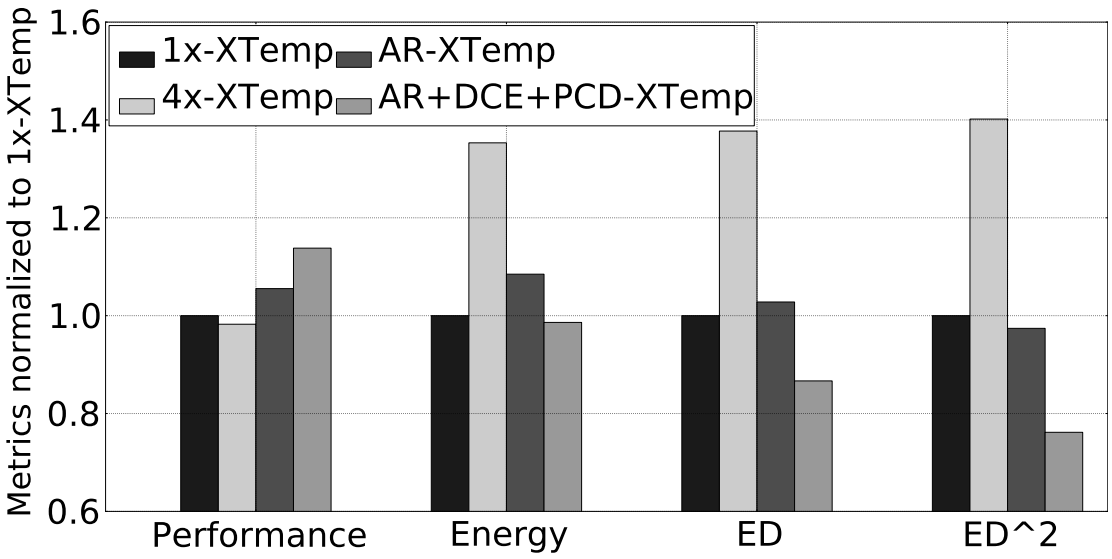

Figure 3.16: Mean performance, energy, energy-delay and energy-delay squared in the extended DRAM temperature range for the 1x, 4x, AR and AR+DCE+PCD configurations, normalized to that of the 1x configuration.

and extended DRAM temperature ranges. In the normal DRAM temperature range, AR+DCE+ PCD improves performance by 8% on average, and reduces energy-delay and energy-delay squared by 7 and 14%, respectively. In the extended DRAM temperature range, AR+DCE+ PCD improves performance by 14% on average, and reduces energy-delay and energy-delay squared by 14 and 24%, respectively. As expected, because of the increase in the volume of refresh operations, the 4x configuration exhibits the highest energy consumption. Thus, our proposed AR+DCE+PCD not only improves performance significantly, it does so by consuming the same amount of energy when compared to the 1x configuration, for an overall improvement in energy efficiency.

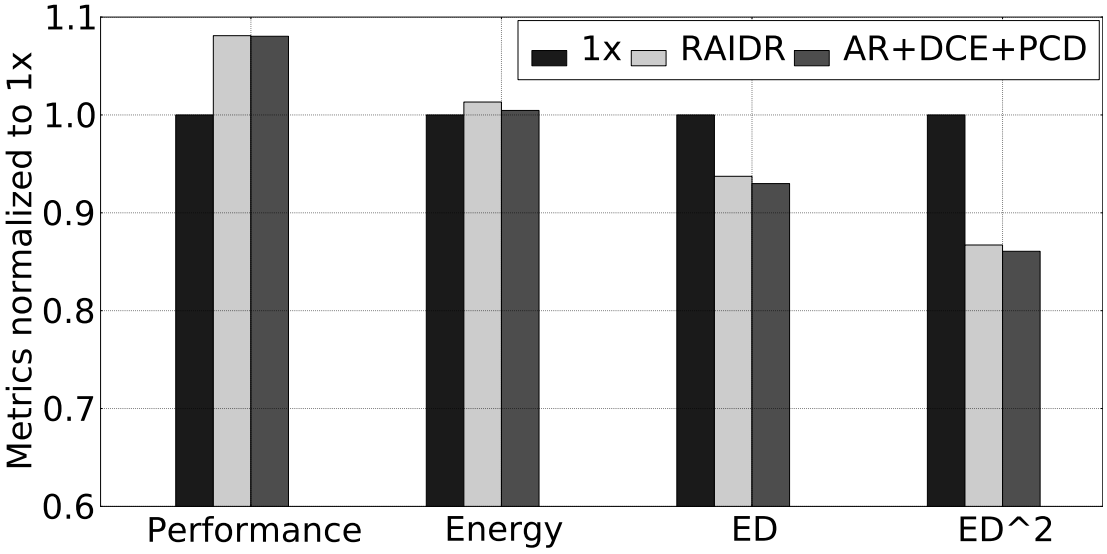

Figure 3.17: Average performance, energy, energy-delay, and energy-delay squared in the normal DRAM temperature range for the 1x, RAIDR and AR+DCE+PCD configurations, normalized to that of the 1x configuration.

3.8.5 Comparison to RAIDR

Figure 3.17 shows the performance, energy, energy delay, and energy delay squared numbers for the the 1x mode, RAIDR [44], and AR+DCE+PCD. RAIDR is a RAS-only refresh implementation that cuts down significantly on the number of refreshes by exploiting inter-cell variation in retention, resulting in improved performance and energy efficiency (Section 3.3). The plot shows that AR +DCE+PCD virtually matches RAIDR in every performance and energy metric. This is significant because it does so (1) without resorting to RAIDR's RAS-only refresh (which bypasses DRAM's auto-refresh feature and requires the controller to identify on the address bus which bank needs to be refreshed at each point in time), and (2) without making any assumptions on retention times of DRAM cells, which may cause reliability issues (Section 3.3).

3.9 Conclusion

Our analysis of DDR4 DRAM's new Fine Granularity Refresh feature shows that there is no one-size-fits-all refresh option across the applications that we have used in our study. This makes our proposed *Adaptive Refresh (AR)* mechanism a simple yet effective way to leverage the best FGR mode in each application and phase within the application.

For high-density DRAM systems, we have identified a phenomenon that we call *command queue seizure*, whereby the memory controller's command queue seizes up because it is full with commands to a rank that is being refreshed. To attack this problem, we have proposed two complementary mechanisms called *Delayed Command Expansion (DCE)* and *Preemptive Command Drain (PCD)* which increase the number of issuable DRAM commands in the scheduler's command queue when a refresh operation is underway.

Once our proposed DCE and PCD mechanisms are in place, DDR4's FGR becomes redundant in most cases, except in highly memory-sensitive applications, where the use of AR does provide some additional benefit. In all, the proposed mechanisms yield significant performance gains with respect to traditional refresh at both normal and extended DRAM operating temperatures.

CHAPTER 4

IMPROVING I/O SCHEDULING FOR FLASH-BASED SOLID STATE DRIVES (SSDS) THROUGH FORESIGHT

Over the past few years computer systems of all types have started integrating flash memory. Initially, because of its small size, low power consumption, and low-cost I/Os per second, flash was a natural fit for embedded devices. As NAND flash scales, flash memory's high density and low cost make it a viable option for desktop and high-end server environments. However, as this scaling continues, endurance of flash chips is projected to drop as well. Therefore, designing flash controllers for modern I/O systems will require more sophisticated scheduling algorithms that not only improve performance and I/Os per second but also encompass other issues like endurance and write amplification.

4.1 Introduction

Modern enterprise-class SSDs are designed to provide high performance by exploiting concurrency. The number of channels in an SSD, the number of flash devices that connect to a single channel, and the internal parallelism within each device have all increased in recent years. Although increased concurrency generally results in higher performance, Wang et al. show that there is a fundamental tradeoff between parallelism and garbage collection overhead due to data migration (known as *write amplification*), which complicates this relationship.

Another trend in SSD designs has been the decreasing endurance of individual flash cells, which has resulted in devices that are increasingly sensitive to wear. This has magnified the importance of wear-leveling algorithms, which

attempt to maximize the lifetime of the system by distributing wear-causing program and erase operations evenly among the physical memory blocks in the SSD. Again because wear-leveling, like garbage collection, involves data migration (and hence, write amplification), these algorithms affect not only the lifetime of an SSD, but also its performance.

Because of the inter-dependencies of the factors that govern SSD performance and lifetime, the ability to meet particular goals for each metric depends in large part on the ability of the SSD controller to schedule and reorder SSD commands (including commands related to data access, garbage collection, and wear leveling) to take advantage of the highest degree of concurrency possible, while attempting to minimize write amplification, and to evenly distribute wear in the system. However, many existing SSD controllers use heuristics for command scheduling that can neither be tailored to suit the requirements of an individual system, nor can they be adapted over time in response to changing runtime conditions.

Recently, Mukundan and Martínez proposed MORSE, a reinforcement learning (RL) based technique for designing self-optimizing DRAM schedulers to target arbitrary objective functions [52]. Reinforcement learning works, in this case, by having the scheduler use the experience it has gained over time by interacting with the system to schedule commands in such a way as to maximize the value given by an objective function. MORSE was shown to provide significant gains in performance, energy efficiency, and fairness, when compared to competing DRAM scheduler designs.

This work builds off the RL-based technique discussed by Mukundan and Martínez, and proposes a framework for developing SSD schedulers that target,

in a unified manner, not only performance, but also domain-specific idiosyncrasies such as garbage collection and wear leveling. Using this framework, we present SSD scheduler designs that target performance and system lifetime, and we present a quantitative case for these designs' superior ability to capitalize on the trend of increasing concurrency in maximizing these targets, when compared to existing SSD controller designs.

The remainder of this chapter is as follows: in section 4.2, we describe SSD operation, and highlight some of the factors that complicate SSD scheduling. In section 4.4, we propose our technique for designing SSD schedulers. Finally, in section 4.5 we discuss how we evaluated our technique, and in section 4.6, we show that our RL-based SSD controller compares favorably to existing state of the art designs.

4.2 Background

4.2.1 Flash Memory Overview

In this section, we discuss the basics of NAND flash-based SSDs. The numerical values given below are intended to represent an enterprise-like design, and are the parameters we use in our evaluation.

Flash Memory Organization:

Inside our NAND flash-based SSD, an SSD controller manages eight independent channels. Each channel is connected to one NAND flash package, which

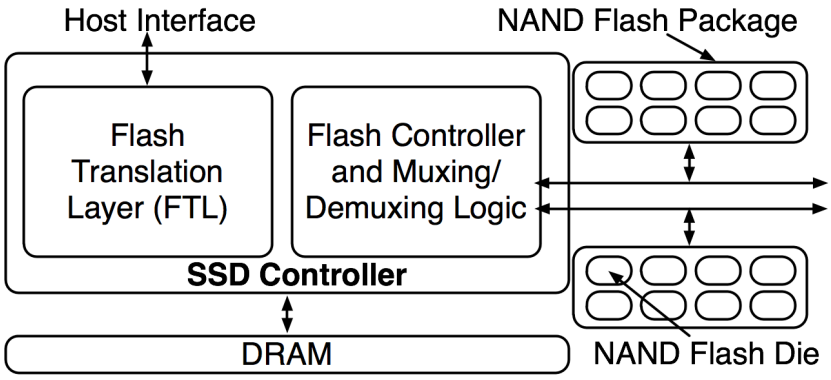

Figure 4.1: Block diagram of SSD system. The SSD controller contains the flash translation layer which is responsible for issuing flash commands and performing maintenance operations on the SSD dies. In this figure we see two flash packages communicating with the SSD controller using independent channels or buses. Each flash package also has eight independent flash dies.

includes eight dies. (It is possible for multiple packages to share a channel, which increases the level of concurrency.) Dies are further subdivided into two planes, each of which contains 2,048 blocks. Finally, a block consists of 256 16KB pages. Figure 4.1 shows an SSD system with two independent channels. Each channel is connected to a NAND flash package with eight dies.

Flash Memory Operations:

Flash memory devices are capable of performing the following basic operations:

- **Read**: Transfer the data stored at a given page address from the flash back to the host.
- **Program (write)**: Store the data from the host at the given page address. This involves selectively clearing certain bits in a free (erased) page. The choice of which logical page to use for new writes, known as the *write-*

placement strategy, can be an important factor in determining both the performance and the endurance of the system.

- **Erase**: Set all of the bits in a physical block to '1.' This puts the block in the free state and capable of receiving writes. When determining which blocks to erase, and when, it is important to consider that blocks can only be written to a finite number of times before wearing out. (The *endurance* of a flash block is on the order of thousands to tens of thousands of erases, depending on the technology.)

Note that in the operations described above, writes can only be performed on pages that have been erased. This behavior, known as *out-of-place updating*, complicates the relationship between the SSD and the host (which is generally configured for accessing block devices with *in-place updating*, such as hard disks). Also note that writes operate on an individual page, while erases affect an entire block. This makes more challenging the problem of keeping enough pages free to receive future writes. These issues will be addressed by the flash translation layer, as discussed below in Section 4.2.1.

Garbage Collection:

Because SSDs require a block to be erased before any pages can be written to it, a process known as garbage collection is required to ensure that there is sufficient supply of erased blocks in the system at any given time. Garbage collection first involves migrating all of the valid pages in a particular block into a separate free block, which is accomplished by reading the valid data from the old block, and then writing it to the new block (known respectively as *read relocate* and *write relocate* commands. After migration the first block is marked as invalid,

and therefore eligible for erasure. Collectively, read relocate, write relocate, and erase commands are referred as *maintenance* operations.

Wear Leveling

Flash memory cells can only be erased a finite number of times before they wear out. These effects are generally expressed as a maximum number of erases to a given block that can be performed, before that block is considered unreliable; these counts are generally in the tens of thousands for the type of enterprise-grade SLC flash we are considering. Wear leveling refers to the attempt to distribute wear evenly among the blocks in the system–during write placement and garbage collection–in order to maximize system lifetime. Wear-leveling algorithms may also decide to move static data to heavily-worn blocks, to help spread the wear to other blocks.

Flash Page States

A Flash page can be in one of three states at any given time:

- **Valid**: page contains valid data
- **Invalid**: page contains invalid (stale) data
- **Free**: page contains no data

A *valid* page becomes *invalid* if the logical address it maps to is re-written to another physical location. An *invalid* page becomes *free* when the block it resides in gets erased.

Flash Block States

Figure 4.2 illustrates the possible states for each block: *free* blocks have only free (erased) pages, *active free* blocks have some free pages and some pages that are valid/invalid, *active* blocks have no free pages, but still have some valid data, and finally *inactive* blocks contain only invalid pages.

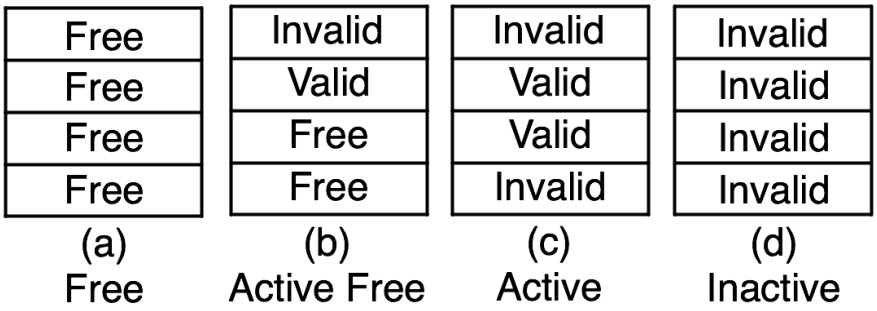

Figure 4.2: A flash block can be in one of four states. A free block contains all pages in erased (clean) state. An active-free block contains at least one free page. An active block contains valid and invalid pages, but no free pages and finally an inactive block contains only invalid pages.

The finite state machine in Figure 4.3 shows how a given block transitions among these states. When the SSD is initialized, all blocks begin as *free* blocks. Once a free block is written to, it transitions to the *active free* state, where it will remain as long as it contains *free* pages to write to. After the last *free* page has been written to, the block transitions to the *active* state. It will remain in this state until it is chosen for garbage collection, at which point it will move to the *inactive* state. Alternatively, subsequent writes can invalidate all its pages, making this state change partially data dependent. Finally, *inactive* blocks transition to the *free* state when they are erased. The SSD controller can explicitly choose which *free* blocks to write to, which *active* blocks to garbage collect, and which *inactive* blocks to erase, so it has nearly complete control over the ways in which blocks

transition between these states.

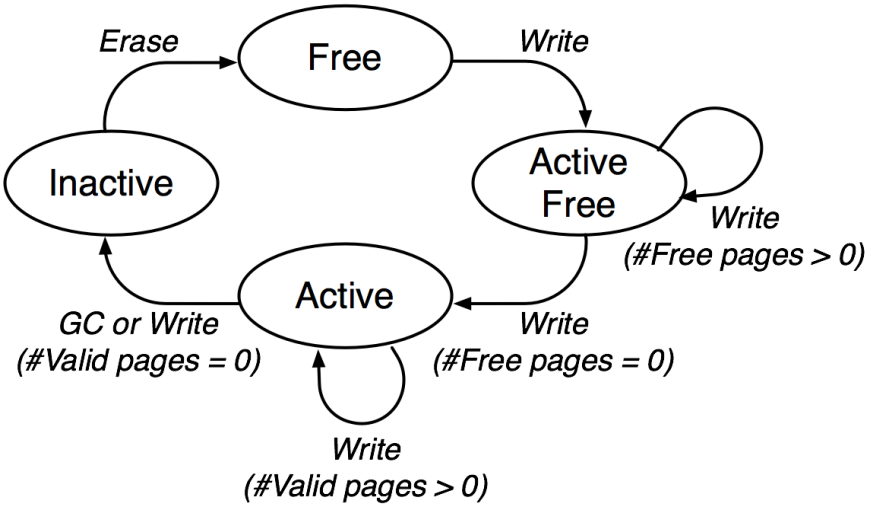

Figure 4.3: Finite state machine indicating the state transitions of a block. Initially all blocks start in free state. A write to a block makes it active free. A write to the last free page in an active free block transitions it to active state. A block will remain in active state as long as it has at least one valid page. When the last valid page has been invalidated because of subsequent writes or garbage collection, it moves to inactive state, where it remains until it gets erased. This transitions it back to the free state.

Flash Translation Layer (FTL)

The issues of out-of-place updating and program/erase granularity mismatch are addressed by a special layer called the *flash translation layer* (FTL). The FTL is primarily responsible for maintaining the physical page location for each logical page in the system. As updates for a given logical address arrive from the host, the FTL selects a clean page to write the new data to, and then changes the translation table entry for that logical address to point to the physical address of the new page.

There are several common alternatives for storing this translation information, ranging from *block-mapped*, which forces writes to occur to the physical page with the same fixed offset as the logical address (saving memory in the translation table), to *page-mapped*, which is fully associative and allows a logical page to be stored in any physical page, but requires a large amount of memory for the translation table. Many proposals employ a hybrid mechanism, known as *log-mapped* translation, but in order to realize the full possibilities of our reinforcement learning framework, we chose a page-mapped implementation. (The memory requirements for the translation table in our design are discussed in Section 4.3.)

In addition to maintaining translation information, the FTL is responsible for keeping a ready supply of free pages for new writes with garbage collection, and for doing so in such a way as to help maximize block endurance and prolong system lifetime through wear leveling. Write placement, garbage collection, and wear leveling, are complex, multi-dimensional problems that impact various system characteristics such as the performance and the endurance of the SSD. For example, an attempt to reduce the latency of writes through aggressive garbage collection, can cause a write amplification problem, which both hurts performance by delaying user reads and writes from being issued, and increases the wear of the system. A good FTL, then, is a key factor in designing an SSD for demanding enterprise environments.

4.2.2 Basics of Reinforcement Learning

Reinforcement learning (RL) involves training an autonomous *agent* to select an action at each point in time, with the goal of maximizing some reward over the long-term, by interacting with, and learning from a probabilistic environment [67]. Importantly, what makes RL distinct from greedy optimization techniques, is that the agent does not simply select the actions that yield the greatest *immediate* reward, but rather, it tries to choose actions that will maximize the *cumulative* reward. Thus, for some given time step, it may pick actions that do not appear to be beneficial in the near-term–and in fact, may even seem to detract from the near-term reward–if it predicts that doing so will help to escape local optima.

One of the keys to enabling this kind of foresight and planning is known as *exploration*. Exploration is the process of choosing a random action for a given time step, observing the results, and updating the agent's internal model of the environment accordingly. This helps the agent to make better predictions in the future, because it will have more experience to draw on. Exploration is the dual of the agent's other primary mode: *exploitation*, which is the process of choosing the particular action that is predicted to maximize long-term reward.

An RL agent must balance exploration and exploitation carefully; too much exploration causes the agent to take a long time to reach the optimal policy, while too little exploration may cause the agent to be stuck with a sub-optimal policy. Furthermore, the agent can never stop exploring completely, because it must constantly adapt its policy over time to changes in the environment, such as when entering a new program phase. To achieve this balance, the scheduler implements an exploration mechanism known as ϵ-greedy action selection, in

which the scheduler sometimes picks a random (legal) action with a small probability ϵ.

4.2.3 RL Model

Every RL model comprises three basic components:

- A state representation that maps the current conditions of the environment to one in a set of states
- A set of actions that can be performed by the RL agent
- A reward function to determine the credit associated with transitioning among states after performing a given action

Each of these components can be represented in an SSD–where the RL agent is the command scheduler in the FTL, and the system as a whole is the environment–as follows: the vectorized list of relevant system attributes is the state-representation (the process of selecting relevant attributes for inclusion in the state-representation is called *feature selection*, and will be discussed further in Section 4.4.1). The SSD commands in the queue at any particular time are the set of actions that the RL agent must select from. The specific reward function depends on the weighted priorities of the target metrics being optimized (performance, in this case).

RL Operation

At every time step (a single SSD scheduling quantum), the agent must observe the state of the system, and choose the action from among the ready commands in the queue that it predicts will obtain the maximum reward from the reward function. It then performs that action, observes the resulting change in state, and determines the actual credit (or blame) associated with its decision. A common technique for credit assignment in RL problems is called *Q-Learning*, which attempts to quantify the cumulative reward (referred to as the *Q-value* that will result from taking action a, while in state s, and then continuing following policy π, represented as $Q_\pi(s,a)$. The RL agent tries to approximate the optimal policy π^* by filling in the *Q-value matrix*, which consists of the values of $Q_{\pi^*}(s,a)$ for every state-action pair, and updating these Q-values as the agent gains experience over time, according to the SARSA update rule as discussed in [67].

4.3 Architectural Support for SSD Controller

4.3.1 Baseline Scheduler

Scheduling is a complex problem that is provably NP-hard. While more sophisticated scheduling algorithms can provide additional performance benefits over a simpler approach, they do so at the cost of increased complexity. The more complex the scheduling scheme, the more time it takes to schedule a command. In many high-end server and enterprise applications (such as the OLTP-like workloads we target), requests from the host arrive at the SSD at a higher

rate than individual commands can be processed. These incoming requests are queued by the SSD upon arrival, until they can be issued by the controller as commands to the flash devices. Nam et al. propose Ozone (O3), and discuss the tradeoffs associated with scheduling these requests in order, vs. scheduling them out of order[55]. They show that out-of-order scheduling allows for better utilization of the inherent parallelism in the multi-level hierarchical organization of the flash memory. To that end, we will be comparing our proposed technique against the decoupled out-of-order scheduling algorithm in Ozone (O3). The O3 scheduling algorithm first prioritizes ready commands over commands that are not ready, and then older commands over younger commands. A command is considered "ready" according to the following rules:

- A *read* command is "ready" if the flash channel, die, and plane that holds the logical address to be read are not currently locked in another operation.
- A *write* command is "ready" if an *active-free* block exists, and the corresponding flash channel, die, and plane are not locked in another operation.
- A *erase* command is "ready" if the flash channel, die, and plane corresponding to the block that needs to be erased are not locked in another operation.

4.3.2 Hardware Support for SSD Controller

In order to realize the benefits of out-of-order scheduling, we need a controller that can efficiently support specialized operations without compromising the scheduling quantum. Therefore, while an SSD controller can be implemented

either in hardware, or in software, we have chosen to focus our efforts on a hardware approach because it can perform these operations in parallel effectively.

Two of the primary functions that an SSD controller performs are garbage collection and wear leveling. Generally, the controller picks the block with the lowest number of valid pages for garbage collection. Similarly, for wear leveling, the block with the lowest erasure count is chosen. Because the order in which blocks change state is partially data dependent (section 4.2.1), the controller will not know ahead of time the best candidate for these operations without the overhead of keeping lists of sorted blocks based on erasure counts and valid pages. Sorting a list of blocks on the fly would dramatically hurt our scheduling quantum. Instead, we chose a binning approach that maintains a dynamically pseudo-sorted list, so that blocks with low erasure counts or numbers of valid pages can always be quickly found. An added benefit is that the same hardware can also be leveraged to find low erasure count blocks for write placement.

SSD Maintenance Using Bins:

The number of bins and the granularity of binning are parameters that can be tuned, depending on the system. For our experiments, we have found the setup shown in Table 4.1 to provide a good balance between accuracy and complexity[1].

To implement these bins, we use a hardware implementation of a doubly-linked list. The SSD controller uses registers for each of the three bins inside

[1]We have observed that binning provides similar accuracy when compared to sorting blocks based on erases or valid pages.

each list to store the *head* and *tail* pointers (physical addresses)). It must also keep track of the *next* and *previous* pointers for each block within a bin.

With the above framework, choosing a block to garbage collect, for example, involves picking the block at the *head* of *bin 1* in the *active* list. Similarly, to choose a block to write to, the controller uses the block indicated by the *head* pointer for *bin 1* in the *free* list. When a block transitions from one state to another, the controller must bin it according to its properties, and update the corresponding *head*, *tail*, *next* and *previous* pointers. (Section 4.3.3 discusses the latency of these updates during a scheduling quantum in detail.) With these bins in place, the SSD controller's write placement, garbage collection, and wear leveling decisions are dramatically simplified, and require no on-the-fly sorting.

Table 4.2 describes the hardware structures required by the SSD controller to perform scheduling and maintenance. Some modern SSD controllers have a dedicated DRAM of 512MB or more [13]. We leverage this DRAM to store the larger hardware structures required to keep track of the state of the SSD including the properties of the blocks. Those structures that are accessed frequently are stored in SRAM. We sized these structures to fit within 1MB of SRAM to be comparable with the SRAM caches in the Atom-like microprocessors that are often used to implement controllers in SSDs.

4.3.3 Scheduling in Hardware:

With these hardware structures in mind, we briefly describe how the O3 algorithm works in hardware. We will discuss the extensions required for the RL-based scheduler in section 4.4.2, but the procedures discussed in this section

Table 4.1: Binning criteria for each of the block lists. For example, consider an active block whose erase count is 30% of the maximum number of erases. If this block were to become inactive, it would be inserted at the end of bin 2 in the inactive list

State	**Criteria**	**Bin 1**	**Bin 2**	**Bin 3**
Free	# Erases	0%-25%	25%-50%	50%-100%
Inactive	# Erases	0%-25%	25%-50%	50%-100%
Active	# Valid Pages	0%-25%	25%-50%	50%-100%

Table 4.2: Hardware structures used in Binning

Structure	**Description**	**Size**[2]	**Location**
Translation Table (TT)	Holds Logical to physical page address mapping	48MB	DRAM
Block Status Table (BST)	Stores state, # valid pages and # erases for all blocks	12KB/die	SRAM
Block FSM Table (BFSMT)	Tracks state, bin number, next and previous pointers for each block	16KB/die	DRAM
FSM Pointer Table (FSMPT)	Stores head and tail pointers for the bins in each list	27 bytes/die	SRAM
Active Free Register (AFR)	Holds physical address of next free page in die	18 bits/die	Register
Channel Free Register (CFR)	Stores bit mask that indicates channel-locked status	8 bits	Register
Die Free Register (DFR)	Stores bit mask that indicates die-locked status	64 bits	Register

are common to both schedulers. Each scheduling quantum can be divided into four logical phases as indicated in figure 4.4:

1. Choose ready command based on scheduling algorithm

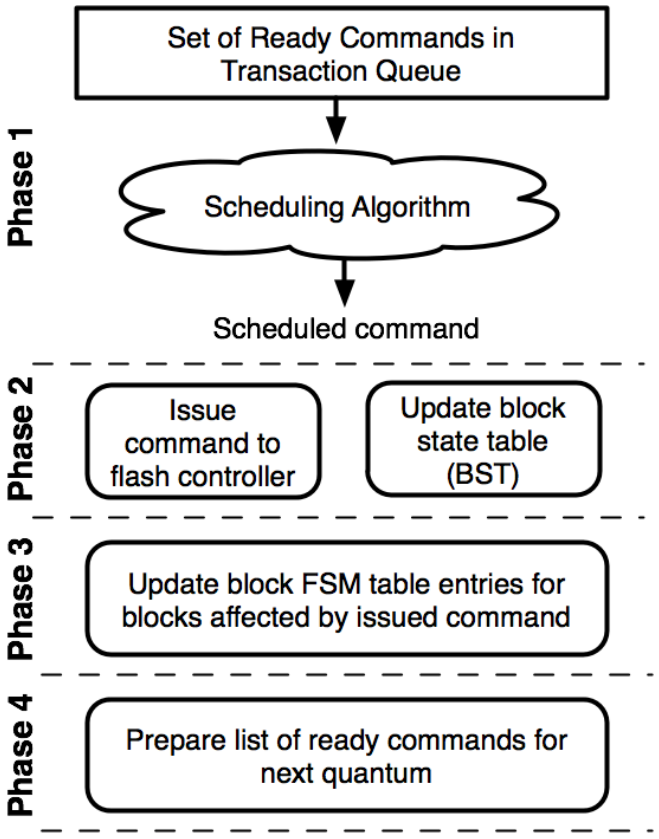

Figure 4.4: The four phases of a scheduling quantum. The first phase involves choosing one ready command from to issue based on the scheduling algorithm. In the next two phases, the command is issued, and the state of the SSD, the state machine and the blocks are updated. Finally, in the last stage, we pick the ready commands that will be available for the next scheduling quantum.

2. Issue ready command and update the block state table (BST)

3. Update the finite state machine for the blocks (BFSMT)

4. Prepare list of ready events for the next quantum

Choose ready command based on scheduling algorithm:

At the beginning of each scheduling quantum, the controller already has a list of commands that are ready for issue (this list is computed at the end of the previous quantum), and the task of the scheduler is to choose one of these commands to issue. In the case of O3, this is a simple decision: the controller chooses the ready command that arrived first. We discuss how the RL controller picks a ready command in section 4.4.

Issue and Update BST:

In the case when the scheduler picks a write command to be issued, we start by reading the TT to get the old physical page address where the data was previously stored (this is needed to invalidate the old page). In parallel, we read the AFR of the appropriate die to get the new physical page address to write to. Next, we update the TT to reflect the new physical location of the data. Since a write event changes the state and the properties of a block, we must update the BST as well. Additionally, the AFR needs to be changed to point to the next free page. On the other hand, when the scheduler chooses a read command, we already know where the data resides (from the previous scheduling quantum), and hence it can be issued immediately.

Once a command gets issued, it causes the channels and dies to be locked for a certain amount of time (depending on what was issued). The critical path in this phase requires going to DRAM to read and update the TT. All other structures are in SRAM and hence do not affect the latency of the critical path. Conservatively assuming that a DRAM access takes 100 ns [4], this phase takes

200 ns to complete.

Update the state machine for the blocks (BFSMT):

Issuing a read event does not cause any updates to the state machine, but issuing a write event can cause blocks to change their bins and potentially their states. For example, if the write event is to the last free page in an *active free* block, this causes the block to move to *active* state. Additionally, it also gets a new bin number depending on the number of pages containing valid data. The write event could invalidate pages in other blocks, which could potentially transition these blocks to another bin in the active state. The write event could also cause the old block to transition to an *invalid* state, at which point it gets a new bin number depending on the number of times it has been erased.

Because these bins simulate a doubly linked list in hardware, the corresponding, head, tail, next and previous pointers must be updated to account for these changes. In the worst case, there are updates to five different blocks in the BFSMT. This happens when the following occur: (a) We write to the last free page the current *active free* block, which makes it *active*. (b) We invalidate a page in an old *active* block, which causes it to transition bins. The critical path for this phase requires one DRAM read to the BFSMT for the new block, and possibly a maximum of five DRAM writes to update the BFSMT. Therefore, in the worst case this phase takes 600 ns.

Get ready events for the next quantum:

To exploit out-of-order scheduling, we need a sufficiently large set of ready commands to pick from. Empirically we determine that for a transaction queue depth of 128 entries, a scheduling window of 64 is sufficient to guarantee at least one ready command per scheduling quantum. In this scheduling window, We need to perform a dependency check on these 64 commands to to eliminate data hazards. This dependency check logic requires 2016 logical address comparisons, which takes 464 ns (Perri and Corsonello [56]) .

Next, for each read event in this subset, we query the TT to get the physical page address. In the worst cases scenario, when we run applications that are read centric (like sequential read and random read workloads), this involves up to 64 DRAM reads to the TT, which will take 6400 ns. We then read the CFR and DFR to get free channels and dies that can accept events. The total latency of this phase is 6900 ns. With this information, at the end of the scheduling quantum, we know the set of ready events that can be issued in the quantum. Summing up the latency of each of the phases gives us the scheduling quantum, which adds up to: 7700 ns.

4.4 RL-based Self-Optimizing SSD Schedulers

4.4.1 Design of the RL-based SSD Scheduler

We now examine in more detail the components of our RL-based SSD scheduler as discussed in section 4.2.2 in light of the additional background on flash and

scheduling.

RL Actions:

The commands that our RL-based scheduler may encounter in the queue are as follows:

- **Read:** Read user specified flash page
- **Read Relocate:** Similar to a read operation, but the request is specified by the garbage collection engine.
- **Write** Write user specified flash page to a free page.
- **Write Relocate:** Similar to a write operation, but the request is specified by the garbage collection engine.
- **Erase:** Erase specified block.
- **Wait-Before-Erase (WEB):** This action delays the scheduler from issuing an erase command.

State Representation:

Recall that at every time step the RL scheduler has the capability of sensing its environment and identifying its state via a set of attributes. Because it is not possible to represent all aspects of the system state in hardware and still be able to meet area, timing and power requirements, it is crucial to pick a representative set of state attributes that can sufficiently characterize the environment, a process known as *feature selection*. We chose to perform feature selection using a heuristic search technique known as genetic algorithms.

Reward Function:

The absolute and relative values of the immediate rewards associated with each action are critical component of an RL system that can influence the rate of learning. For a simplistic RL scheduler, the rewards can be manually assigned based on the designer's expert intuition. However, for a sophisticated design whose objective functions seek to optimize complex and interrelated metrics such as wear and write amplification, an appropriate immediate reward function is not at all evident. Furthermore, the rewards that may suit one application (or program phase), may not work well for another. Therefore, just as with feature selection, we use genetic algorithms to evolve a suitable set of immediate rewards for our target applications.

Deriving the State Representation and Reward Function using Genetic Algorithms:

Genetic algorithms (GAs) are a class of heuristic techniques for searching a multi-dimensional design space [51], that are based on evolutionary processes. GAs help to search for the optimal *candidate* from a *population*, where candidates are represented as a vector of features. These features can be binary values, integers, or real numbers, depending on the application, but in our case, these features are a combination of real numbers, representing the reward values for each action as discussed in section 4.4.1, and binary values, which indicate whether a particular characteristic of the system is to be used or not used as an attribute in the state representation as discussed in section 4.4.1. A set of reward values and state attributes is referred to as a *policy*.

In order to use GAs to find the optimal candidate policy, we start with a randomly generated initial population of 100 policies. That is, the feature values for each of the policies in the initial population is randomly generated. Each of the policies in the initial population is used to simulate the SSD for several applications in the *testing set*. The results of each of the simulations is ranked according to a *fitness function*, which is used to encode the designers' priorities into the RL agent. For our experiments, we choose to use performance (throughput) as our fitness function.

Once the fitness values have been determined, the policies are randomly assigned to participate in *tournaments* with each other, where the policy with the highest fitness in the tournament wins. (The reason for the random selection of tournament participants is to avoid the automatic elimination of all lower-performing policies in favor of higher-performing ones, which could lead to getting trapped in local optima.) Finally, the winners of these tournaments undergo transformations as part of the evolutionary process to create the next generation of candidate policies. Examples of these transformations include *crossover*, in which part of the vector of features for one policy is swapped with part of another policy, and *mutation*, wherein certain feature values for a policy are randomly changed. These transformations help to more rapidly explore various regions in the design space.

This process, of simulating the policies in a population, computing their fitness values, holding tournaments, and transforming the tournament winners, is continued iteratively as time allows, or until the improvement in fitness from one generation to the next is sufficiently small. At the end of this iterative process, the policies seen in all of the generations are ranked according to their

fitness values, and the policy with the highest fitness is chosen for the operating mode associated with that fitness function.

Results from the GAs

The state attributes for our proposed RL configuration (which uses the baseline garbage collection engine) when co-evolving attributes and reward function are:

1. Number of reads in the transaction queue for the die under consideration.
2. Total number of reads in the transaction queue.
3. Number of reads in the transaction queue for the plane under consideration.
4. The timestamp of the the most recently issued read for the die under consideration.
5. Number of valid pages in a block.

The individual rewards that were obtained are shown in Table 4.3:

Table 4.3: Individual rewards obtained from the GAs for RL.

Action	**Reward**
Read	0.38
Write	3.15
Relocate Read	-3.36
Relocate Write	4.23

The immediate reward for reads is lower than that for writes. It would be shortsighted to conclude from this that somehow writes are more important than reads in this design, as the RL scheduler makes decisions based on state-action pairs (not actions alone) and an expectation of long-term reward (not immediate reward alone). Keep in mind that in TPC-C the majority of operations are reads in any case, and this weighs in the cumulative reward at the end of the run. On the other hand, recall that the write time is about twice as long as the read time, and that means that every write takes up its die for twice as long. This weighs in favor of squeezing writes in order to free up queue entries; in fact, our simulations tell us that the command queue is often very highly occupied.

Also noticeable is the fact that relocate reads have a negative reward over user defined reads and writes. This makes sense because, in general, we want to prioritize user defined operations over maintenance operations as they are directly related to the performance of the system. The results later will show that this in fact averts a degradation in die utilization that is evident often in execution traces with the O3 configuration.

Note also that relocate writes cannot be issued before their corresponding relocate read commands have completed (to avoid data hazards). Therefore, the fact that relocate writes have a higher reward than relocate reads does not necessarily mean that write maintenance operations will be prioritized over read maintenance operations. Importantly, we see that relocate writes have a higher reward than user reads and writes. In this context, this makes intuitive sense: once a relocate read has completed, we do not want the corresponding relocate write to be delayed waiting in the queue, as this will adversely affect garbage

collection.

When using the automatic GA process to co-evolve the state attributes and the reward structure for an RL scheduler that includes maintenance operations like garbage collection and wear leveling in its decision making process, we obtain the following state vector and reward function. The state attributes are:

1. Number of reads in the transaction queue for the die under consideration.
2. Number of reads in the transaction queue for the block under consideration.
3. Number of valid pages in the block under consideration.
4. Age of the plane.

Table 4.4: Individual rewards obtained from the GAs for RL with GC.

Action	**Reward**
Read	3.55
Write	2.46
Relocate Read	-3.27
Relocate Write	-2.20
Insert Relocates into Queue	4.40
Erase	-3.17
Wait-Before-Erase (WBE)	-0.63

We see that the reward for inserting maintenance operations into the translation queue is higher than both user reads and writes. It is important to understand that for the RL scheduler in the above configuration garbage collection and wear leveling actions are tightly coupled with scheduling. This means

that it not only determines when to relocate pages and issue block erases, but it also decides which pages to relocate or which blocks to erase. Therefore, the scheduler is proactively deciding to insert relocate operations in the anticipation that the SSD will not reach a point where the number of active blocks becomes sufficiently high and the number of free blocks becomes sufficiently low, that garbage collection must kick in, leading to a delay in issuing user defined reads and writes. However, once inserted, we see that the rewards for the maintenance operations are lower than those for user defined operations. Hence, the scheduler is still prioritizing user events over maintenance events, which is important for improved performance. We also notice that the reward for erases is lower than that for a Wait-Before-Erase (WBE). The scheduler, before deciding to issue an erase, checks to see if doing nothing in that cycle (WBE) is better in the long run. This is because, an erase operation is the most time-consuming flash operation, and it ties up an entire die for close to 3ms. No commands can be sent to that die for the duration of an erase operation. Therefore, it must be issued with caution. Comparing against a WBE action which has a higher reward helps monitor the rate at which the erase operations are issued. Finally, notice that the GA process has also picked an attribute that helps improve the endurance of the SSD. The fourth attribute in the state vector represents the number of times the blocks in a particular plane have been erased. With the help of this information, the scheduler can make intelligent decisions about which blocks to pick so that it can improve the wear leveling of the system.

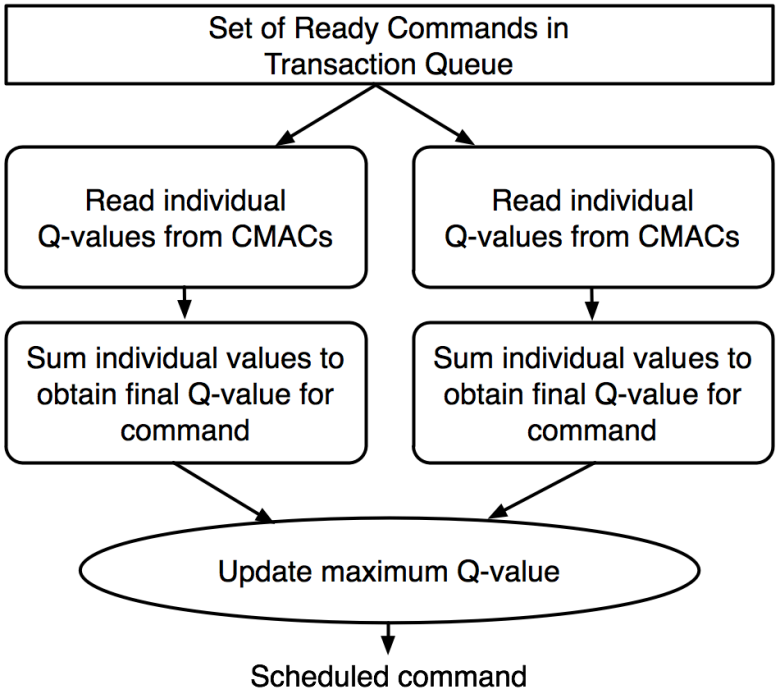

Figure 4.5: The four state RL Q-value estimation pipeline. Each scheduling quantum, the RL pipeline looks at command window of 64 ready commands and picks one command to issue from that set. This takes a total of 132 pipeline cycles. When clocked at a frequency of 1GHz the latency of picking a single command is 132 ns.

4.4.2 Implementation of the RL-based SSD Scheduler

For each scheduling quantum, the RL-based scheduler chooses a command to issue by comparing the Q-values (cumulative long term reward values) of the various ready commands. Figure 4.5 shows the four-stage Q-value estimation pipeline we use in our implementation. As discussed in section 4.3.3, the controller begins each quantum with the list of ready commands. In the first pipeline stage, the RL controller computes the corresponding state-action pairs for each of these ready commands by querying the TT and BST for the relevant physical block addresses and block attributes.

In the next stage of the pipeline, the Q-values for each state-action pair are read out of the CMACs. The indices to read the CMAC tables are calculated using a three step process: first, the upper order bits of the state attributes that were chosen using the GP process described in Section 4.4.1 are concatenated (the quantization granularity of each attribute determines the number of upper order bits that will be used.) Next, the result is XOR-ed with a different constant random number, depending on the command type. Finally, to reduce storage requirements, this XOR-ed value is hashed to generate the index for the CMACs.

In the third pipeline stage, the Q-values from the different CMACs are added up together to obtain the final Q-value for the each state-action pair. Finally, in the last stage of the pipeline, we compare the Q-value computed from the previous stage with the maximum Q-value seen so far, and this maximum value is updated accordingly.

4.4.3 Hardware

The width of the pipeline and number of independent pipes that can operate in parallel can be tuned for different types of systems, based on the number of events that need to be compared, the area, and timing constraints etc. The maximum number of pipeline cycles required to compute and compare the Q-values of *"w"* ready write commands is given by the following equation:

$$cycles = (S + ((w * d)/P))$$

where S is the number of pipeline stages, P indicates the number of parallel

pipes, w indicates the number of ready write commands, and d represents the number of dies explored for write placement. In our implementation we chose to compare a maximum of 64 ready commands per clock cycle using two parallel pipes (32 commands per pipe), and four die options for write placement, for which we describe the timing constraints. Depending on the type of attributes chosen by the GP process, the state-action pairs can be computed by reading the BST and the channel and die registers.

From CACTI [1], we estimate an SRAM read to the BST to be 187 ps. Conservatively assuming that it takes the same time to read the channel and die registers, the first pipeline stage has a clock period of 187 ps. In the second pipeline stage, we assume that the CMACs (SRAM tables) are read out in parallel, and also require 187 ps. Once the CMACs have been read out we, need sixteen 16-bit adders to add up the 32 Q-values from the CMACs (arranged in a four-deep tree). From Wang et al., [70], we estimate the delay of 16-bit adder to be 181 ps. We conservatively estimate the total time required for adding up to 32 Q-values to be close to 1 ns. Finally, the last pipeline stage requires a 16-bit comparator which is estimated to have a delay 230 ps (Perri and Corsonello [56]). We observe that the third pipeline stage is the one in the critical path (1ns), and clock the pipeline at 1 GHz to satisfy its timing constraints.

Using above equation, we estimate that the total number of cycles required to choose a ready command to issue for a set of window of 64 commands is 132 cycles (132 ns).

4.5 Experimental Methodology

We model our system based on high-end enterprise and server based configurations, which have the capabilities of exploiting high degrees of parallelism. All our experiments have been carried out by extending the *FlashSim* simulation environment [38]. Our baseline flash based SSD integrates eight independent channels with the flash translation layer. Each channel supports a single package[3]. Each package has eight independent dual-planed dies. The planes each have 2k blocks and each block supports 64, 16KB flash pages. The latency of a read operation is 25μs, a write operation is 1.3ms and an erase operation is 3.8ms. These numbers were chosen based on discussions with SSD flash vendors in the enterprise domain. The data bus frequency used in our experiments is 200 MHz. Currently, the bus speeds for flash devices are in the range of 40MHz-133MHz, and these values are only expected to increase in future generations [13].

We evaluate a number of configurations as represented in Table 4.5. The O3 configuration performs out-of-order scheduling on a set of ready SSD events [55]. The baseline garbage collection engine starts performing maintenance operations when the number of free blocks falls below a particular threshold. It continues to garbage collect until a certain upper threshold is reached. In our experiments we empirically determine these limits to be 15% and 30% [4]. RL is our proposed reinforcement learning based scheduler that performs out-of-order scheduling on read and write SSD events by incorporating foresight and planning. RL uses the baseline garbage collection engine to perform

[3] Additional packages can be added to independent channels at the cost of bus frequency

[4] We conducted further experiments on different thresholds and found the results to not vary significantly.

maintenance operations. Finally, RL_GC, adds garbage collection into the RL scheduler's decision making process.

Table 4.5: Evaluated Configurations

O3	Out-Of-Order Scheduling proposed Nam et al.[55] + baseline garbage collection
RL	RL based scheduling for read and write events + baseline garbage collection
RL_GC	RL based scheduling for read, write and garbage collection events

We evaluate our proposed schemes on a set of TPC-C application traces obtained from the SNIA IOTTA Repository [3]. The TPC-C benchmark traces was collected at Microsoft using the event tracing for Windows framework. All traces are six-minute long traces collected at various points during periods of steady-state activity [2].

4.6 Evaluation

Figure 4.6 (left) shows the performance obtained when running the TPC-C benchmark traces using the O3, RL and RL-GC configurations, normalized to the performance of O3. RL improves performance over O3 by 10% and RL-GC improves performance over O3 by 12%. As we will see next, RL's and RL-GC's

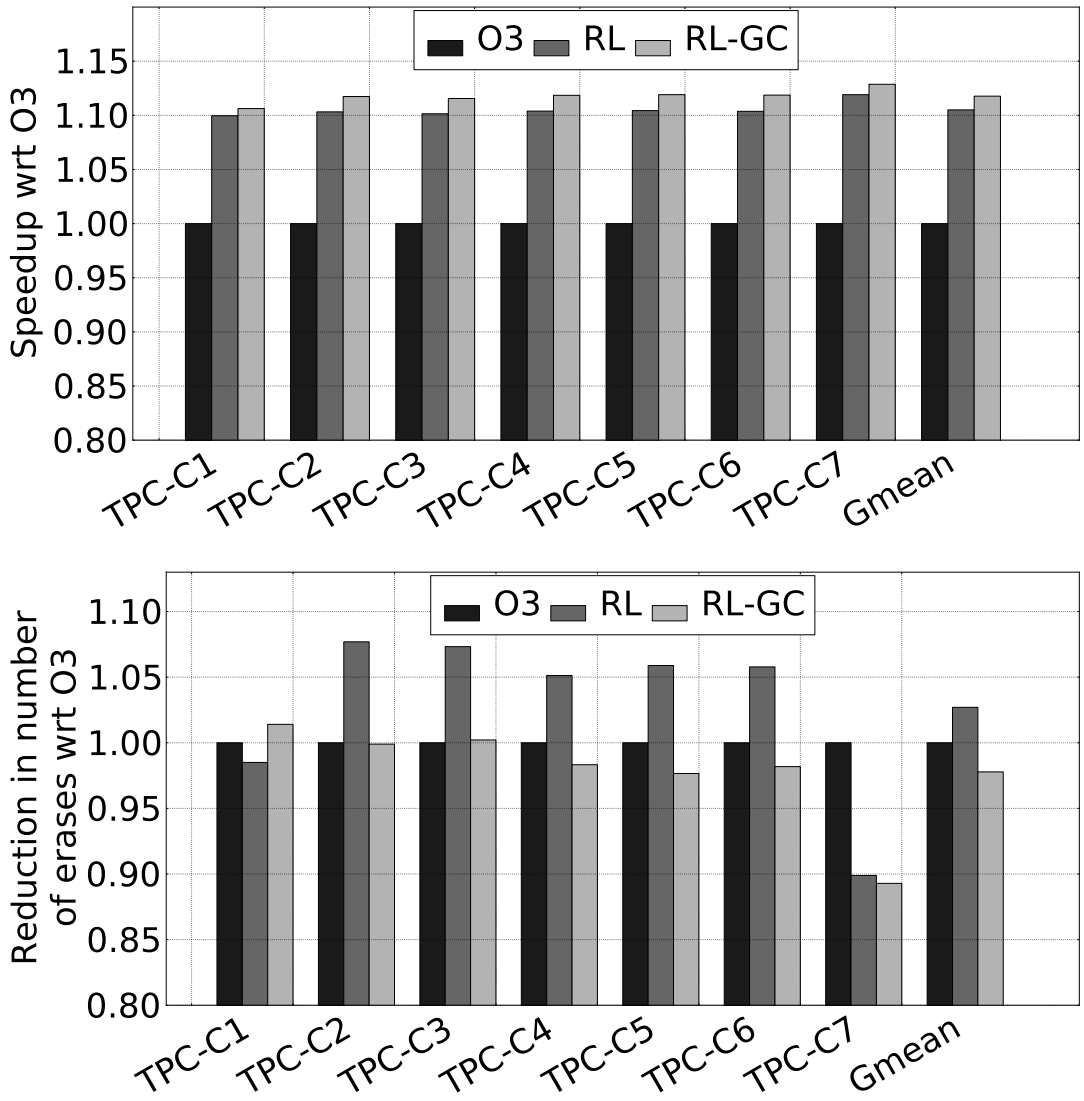

Figure 4.6: Performance (left) and Wear (right) for the TPC-C benchmark traces when running the configurations O3, RL and RL-GC (higher is better).

superiority comes not only from attaining better I/O operations schedules, but also importantly by minimizing the interference of maintenance operations with the schedule.

At the same time, Figure 4.6 (right) shows that the wear sustained by the RL-

GC configuration (as reflected by the reduced number of block erases) is equal to or lower than that of O3, whereas the wear for RL configuration is somewhat higher in many cases. This tell us that the RL configurations are both effective at improving performance, by learning how to best schedule I/O commands at run-time. Moreover, by integrating garbage collection into the scheduler's available actions (RL-GC), the RL configuration *also* improves wear—something the RL configuration that keeps O3's uncoordinated garbage collection mechanism is unable to achieve. Recall also that the RL configuration's objective function does not factor in wear, which results in a reward structure that is agnostic to it, which RL-GC's is not.

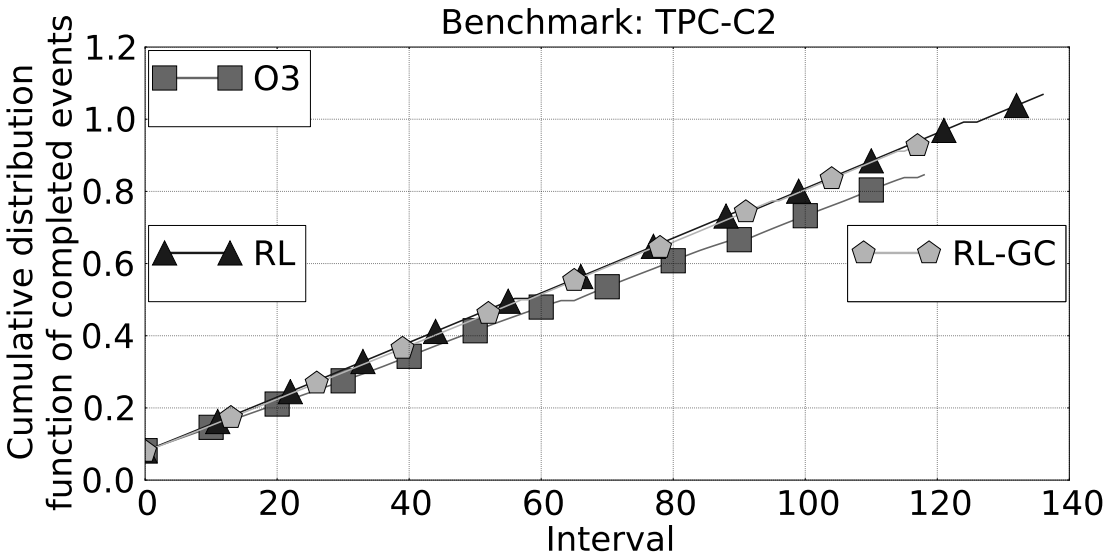

Figure 4.7: Cumulative distribution function of completed events for the benchmark trace TPC-C1, when running the configurations O3, RL and RL-GC (higher is better).

Figure 4.7 shows the cumulative distribution function of completed events for the benchmark TPC-C1 when running the configurations considered in this study. (We verified that this is representative of all the benchmarks studied.)

The plot is reassuring in that it shows how the RL-based configurations attain their gains by sustaining a higher event completion rate than the baseline over the entire run, rather than getting ahead due to some specific, discrete event.

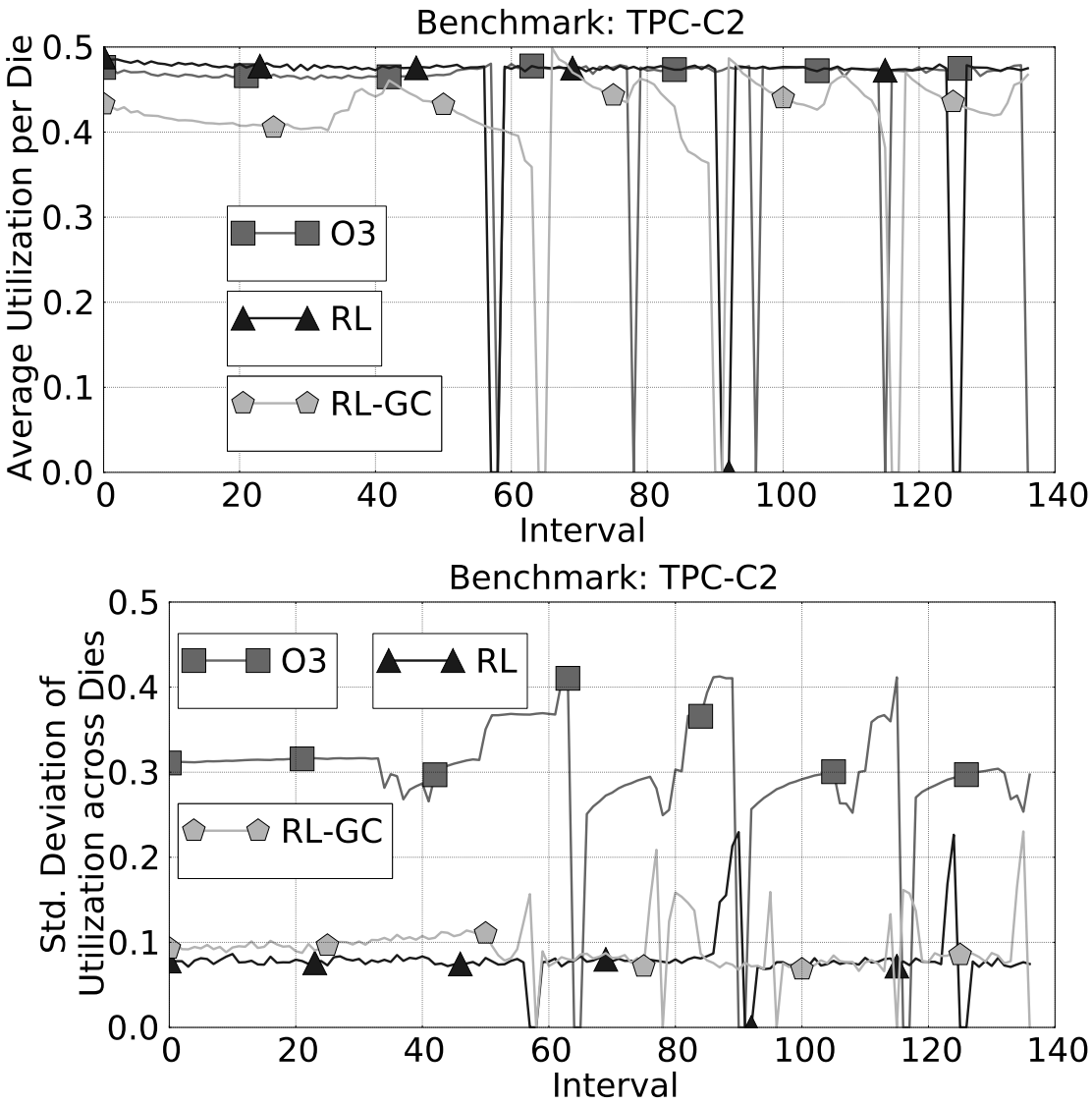

Figure 4.8: Mean (left) and standard deviation (right) of utilzation across dies for the benchmark trace TPC-C1, when running O3, RL and RL-GC

Figure 4.8(left) shows the mean die utilization of each configuration. The utilization dips reflect maintenance phases, which are not counted toward uti-

lization (recall that during these no read/write requests are serviced). The plot shows that, for the RL and RL-GC, utilization is often higher than O3. Two main factors contribute to this:

— First, remember that O3 performs write placement on a round-robin basis, whereas the RL-based configurations assess up to four different candidate dies for every write command. This intelligent write placement exploits the available parallelism more effectively. As a result, the RL configurations on average keep dies busy more often, increasing utilization, and hence performance. Figure 4.8(right) shows the standard deviation of utilization across dies, and shows that intelligent die selection actually results in lower utilization imbalance than the seemingly "even-minded" round-robin policy.

— Second, once the maintenance threshold is reached, the O3 configuration is unable to work around them through changes in its fixed scheduling algorithm, resulting in frequent drops in die utilization. The RL schedulers successfully work around maintenance operations, and succeed at maintaining utilization high wherever there are read/write operations in the command queue.

Overall, these characteristics combined yield higher, more sustained utilization, and thus higher performance.

4.7 Related Work

4.7.1 Flash Translation Layer

Chen et al. propose CAFTL, a content aware flash translation layer that reduces write traffic to flash memory by removing unnecessary duplicate writes and also extends free flash memory by coalescing redundant data is SSDs [16]. DFTL, introduced by Gupta et al. [26], exploits temporal locality to reduce the size of the translation table by caching. Reuse-Aware NFTL (RNFTL) [69] proposes a reuse strategy that utilizes free pages in a un-erased block efficiently to reduce the cost of merge operations. LazyFTL proposed in [49] is a page-mapped FTL that holds small areas of flash memory as update buffers, so that the main page-level mapping table can be updated in a lazy manner. Wu et al. propose an adaptive two-level scheme for the FTL that dynamically adjusts the mapping granularity of logical to physical address blocks [6].

4.7.2 Wear Leveling

Chang et al. [15] propose wear leveling policies that proactively swap hot and cold data between blocks to promote evenness. M-System TrueFFS [48] discusses a round robin policy that allocates new blocks for writes based on erasure counts (static wear leveling) and swaps hot and cold data between young and old blocks (dynamic wear leveling). STMicroelectronics uses a two-level technique that writes new data to free blocks with the fewest erases using methods discussed in [71]. It also swaps hot data with cold data based on the number of write cycles. Kim et al [37] propose a technique that computes a score for

each erasable block in the SSD during garbage collection, and erases the block with the highest score. JFFS [72] and YAFFS [50] propose a turn based selection policy that periodically enables and disables wear leveling during erasure (further elaborated in [61] and [47]). Finally, the dual pool algorithm proposed by Chang et al. in [14] partitions blocks into a hot pool and a cold pool and swaps data if the difference between the erasure cycles of the oldest block in the cold pool and the youngest block in the hot pool reaches a threshold.

4.7.3 Garbage Collection

Kwon et al. introduce FeGC: a scheme with two policies: an efficient block recycling policy based on invalidation history, and a free block management policy to balances the number of erase operations [40]. Ji et al. propose a technique, which exploits data redundancy between main memory and flash memory to minimize garbage collection overheads [34]. Buffer aware garbage collection, proposed by Lee et al. in [42] discuss improvements for block merging and victim block selection to reduce migrations and increase fairness. Han et al. describe an algorithm for flash memory storage systems that predicts future I/O streams, and selects victim blocks based on these predictions [27].

4.8 Conclusion

In this chapter, we have presented a framework for designing SSD controllers that use reinforcement learning techniques to simultaneously target important SSD metrics like performance and endurance. These RL-based controllers are

able to schedule commands using foresight, gained through experience, to maximize concurrency. We have shown that these controllers are able to outperform significantly a state-of-the-art out-of-order scheduler (about 12% on average across several TPC-C benchmarks), and that when the garbage collection mechanism is integrated into the scheduler itself, this is accomplished with no endurance penalty.

CHAPTER 5

CONCLUSION

Scheduling in DRAM based memory systems is provably NP-hard. A number of scheduling algorithms have been proposed in the past, primarily focussing on improving performance. As the storage technology landscape undergoes rapid changes especially because of device scaling, modern memory systems have reached an inflection point. There is a convergence of several trends in DRAM systems, such as the increasing importance of energy consumption, the need for QoS and Fairness etc., Additionally, device scaling has also enabled higher degrees of parallelism, concurrency and density in such systems. Although traditionally computer architects have primarily designed memory systems for improved performance, other objectives like energy efficiency, power and fairness among multi-programmed applications need to be addressed immediately. We find that most existing techniques make fixed static scheduling decisions, handle objective functions individually and lack the capability of long term planning and foresight.

First, this thesis investigates the need for improved scheduling in highly concurrent server-class DRAM systems. We propose MORSE: Multi-Objective Reconfigurable SElf-Optimizing Scheduler: a systematic and general mechanism based on principles of reinforcement learning (RL), that has the capabilities of managing multiple objective functions. We use this general mechanism to present three memory scheduler designs that target performance, energy efficiency, and throughput/fairness. Our first scheduler design – MORSE-P – is able to improve performance of DDR3 memory subsystems by 16% when compared to FR-FCFS and 7% when compared to the performance oriented self-

optimizing scheduler described in [32]. We find that by using the available PwUp/PwDn actions judiciously, MORSE-P is able to derive noticeable benefit by powering down ranks (thereby saving power) without having to close the rows (thereby increasing page hit rate). This allows MORSE-P to improve row buffer locality for the applications which inherently helps improve performance. The appropriate state attribute and reward values help MORSE-P determine when to issue these PwUp/PwDn actions, making them the primary contributors to performance.

Our next scheduler design – MORSE-E – is able to increase energy efficiency by 26%, and reduce energy consumption by 11% in server-class DDR3 systems when compared to a power-aware extension of the FR-FCFS scheduling algorithm. Since our target optimization metric is energy efficiency, the state attributes selected via multi-factor feature selection for MORSE-E have a good mix of features geared towards improving both performance, as well as reducing energy consumption. We find that the rewards obtained using the GA-based automatic reward derivation process tend to favor powering down ranks more frequently, in order to improve energy consumption. It is extremely important to have an efficient power management scheme that puts idle devices into low-power states and activates them at the right time to avoid significant losses in performance. With the help of the energy-aware feature selection and reward function generation process, MORSE-E is able to reduce the average number of active DRAM devices significantly. MORSE-E is able to proactively determine when a rank can be placed in low power mode without hurting pending requests. As a result, ranks stay in this mode for longer on average, resulting in greater energy efficiency.

Our final scheduler design – MORSE-WS – is able to improve weighted speedup of multiprogrammed workloads (a measure of system throughput and fairness) by 8%. It is important to realize that while weighted speedup is a complex metric to measure directly on the field, in our framework it is easy to target at design time through simulations. Once we have derived the features and rewards for a metric, the versatility of RL allows the objective function to be embedded behaviorally on the field, without ever needing to actually measure it. Another interesting aspect we observe is that MORSE-P – our performance oriented scheduler – does equally well when compared to MORSE-WS, even though it was never trained on weighted speedup. But, this is not too suprising, as MORSE-P's objective is more closely aligned with MORSE-WS's. This brings us to the notion of reconfigurability. In a situation where the features are already set in stone, and where the only changes allowed are to a programmable table of immediate rewards, we can envision a solution where the features of a relatable and closely aligned metric can be reused, while deriving the reward function (using offline training) for the new metric, which can then be easily reprogrammed using the OS or firmware updates.

As DRAM scaling continues, it has also introduced several problems that were either easily overlooked or did not manifest in prior memory systems. Chief among these problems is DRAM Refresh overheads, which have been on the rise, especially for high density DRAM memory. Additionally, due to this scaling and increased concurrency, the operating temperature range for DRAMs have also increased. The ability of DRAM cells to be able to retain charge is inversely proportional to temperature. Therefore, at high temperatures, the frequency of refresh operations is doubled, which further exacerbates these overheads.

Second, this thesis understands and analyzes the refresh overheads that surface when moving to high density memory, and quantifies the loss in performance due to these overheads for DDR4 DRAM subsystems. We first evaluate Fine Granularity Refresh (FGR) – a new feature that was introduced in the DDR4 specification to counter increased refresh latencies in high density memory. Our analysis of DDR4 DRAM's new Fine Granularity Refresh feature shows that there is no one-size-fits-all refresh option across applications. We propose Adaptive Refresh (AR), a simple yet effective way to leverage the best FGR mode in each application and phase within the application based on monitoring memory utilization. Adaptive refresh is able to find the best refresh mode for all applications and has very little overhead.

For high-density DRAM systems, we have identified a phenomenon that we call *command queue seizure*, whereby the memory controller's command queue seizes up because it is full with commands to a rank that is being refreshed. To attack this problem, we have proposed two complementary mechanisms called *Delayed Command Expansion (DCE)* and *Preemptive Command Drain (PCD)* which increase the number of issuable DRAM commands in the scheduler's command queue when a refresh operation is underway. Both DCE and PCD provide simple, yet effective forms of foresight and planning as they try to anticipate and prevent a command queue seizure ahead of time. With these new techniques, the performance of high density DDR4 memory improves by 12% with no loss in energy consumption. Once our proposed DCE and PCD mechanisms are in place, we find that DDR4's FGR becomes redundant in most cases, except in highly memory-sensitive applications, where the use of AR does provide some additional benefit. In all, the proposed mechanisms yield significant performance gains with respect to traditional refresh at both normal and extended

DRAM operating temperatures.

Over the past few years, the usage of NAND flash based SSDs is becoming more prevalent, primarily because of it's small size, low power consumption, low cost and due to device scaling. However, as NAND flash continues to scale, much like DRAM based memory systems, I/O systems are faced with similar problems. I/O scheduling is also provably NP-hard and existing I/O controllers lack coordinated management of various objective functions and are unable to understand the long-term consequences of their scheduling decisions. Therefore, finally, this thesis tackles the problem of I/O scheduling in NAND-Flash based SSDs, by leveraging the reinforcement learning based multi-objective reconfigurable framework used for designing self-optimizing schedulers, introduced in the first part of this thesis. We propose an RL based SSD controller that manages write-placement, garbage collection and wear leveling synergistically using foresight and long-term planning. We also provide a methodology for performing the above operations in effectively in hardware by using the concept of binning. The self optimizing SSD controller is able to improve the performance of I/O systems by 12%. There are two main factors that can be attributed to this increased performance. First, the self-optimizing SSD controller performs write placement after assessing up to four different dies, but current systems use a round robin based policy. This intelligent write placement exploits the available parallelism more effectively. As a result, the RL configurations on average keep dies busy more often, increasing utilization, and hence performance. Second, the RL scheduler successfully works around maintenance operations, and succeeds at maintaining high utilization whenever there are read/write operations in the command queue. Overall, these characteristics combined yield higher, more sustained utilization, and thus higher performance.

We believe this work provides attractive techniques to improve various aspects of memory and I/O subsystems, especially in the wake of device scaling. More importantly, it is likely to provide useful insights that can be leveraged in other storage technology domains like PCM, MRAM etc. as well.

BIBLIOGRAPHY

[1] Cacti 6.5. http://quid.hpl.hp.com:9081/cacti/.

[2] Microsoft eneterprise traces. http://iotta.snia.org/traces/130.

[3] Snia iotta repository. http://iotta.snia.org/.

[4] 1gb ddr3 sdram component data sheet: Mt41j128m8, March 2006. http://download.micron.com/pdf/datasheets/dram/ddr3/1Gb_DDR3_SDRAM.pdf.

[5] 2gb ddr3 sdram component data sheet: Mt41j512m4, March 2006. http://download.micron.com/pdf/datasheets/dram/ddr3/2Gb_DDR3_SDRAM.pdf.

[6] An Adaptive Two-Level Management for the Flash Translation Layer in Embedded Systems. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '06, 2006.

[7] Technical note tn-41-01: Calculating memory system power for ddr3, June 2006. http://download.micron.com/pdf/technotes/ddr3/TN4101pdf.

[8] Ddr3 sdram rdimm features, March 2009. http://download.micron.com/pdf/datasheets/modules/ddr3/js-z-s72c1g_2gx72pz.pdf.

[9] Uksong Kang . 8 gb 3-d ddr3 dram using through-silicon-via technology for quasi-non-volatile dram. In *IEE Journal of Solid State Circuits*, 2010.

[10] ASHRAE TEchnical Committee. 2011 Thermal Guidelines for Data Processing Environments Expanded Data Center Classes and Usage Guidance. http://www.eni.com/green-data-center/it_IT/static/pdf/ASHRAE_1.pdf.

[11] V. Aslot and R. Eigenmann. Quantitative performance analysis of the SPEC OMPM2001 benchmarks. *Scientific Programming*, 11(2):105–124, 2003.

[12] D. H. Bailey et al. NAS parallel benchmarks. Technical report, NASA Ames Research Center, March 1994. Tech. Rep. RNR-94-007.

[13] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In *ASPLOS*, 2009.

[14] L.P. Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *SAC*, 2007.

[15] L.P. Chang and T.W. Kuo. Efficient management of large-scale flash-memory storage systems with resource conservation. In *ACM Transactions on Storage*, 2005.

[16] Feng Chen, Tian Luo, and Xiaodong Zhang. CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives. In *Proceedings of the 9th USENIX conference on File and storage technologies*, FAST '11, 2011.

[17] V. Cuppu and B. Jacob. Concurrency, latency, or system overhead: Which has the largest impact on uniprocessor dram-system performance? In *ISCA*, 2001.

[18] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A performance comparison of contemporary dram architectures. In *ISCA*, 1999.

[19] K. Diefendorff. Sony's emotionally charged chip: Killer floating-point 'emotion engine' to power playstation 2000. *Microprocessor Report*, 13(5):1–11, 1999.

[20] M. Eiblmaier, R. Mao, and X. Wang. Power management for main memory with access latency control. In *FeBID*, 2009.

[21] B. Sinharoy et al. IBM POWER7 multicore server processor. *IBM Journal of Research and Technology*, 55(3):1–29, 2011.

[22] Kyomin Sohn et al. A 1.2v 30nm 3.2gb/s/pin 4gb ddr4 sdram with dual-error detection and pvt-tolerant data-fetch scheme. In *ISSCC*, 2012.

[23] X. Fan, C. Ellis, and A. R. Lebeck. Memory controller policies for dram power management. In *ISPLED*, 2001.

[24] X. Fan, C. Ellis, and A. R. Lebeck. The synergy between power-aware memory systems and processor voltage scaling. In *PACS*, 2003.

[25] Mrinmoy Ghosh and Hsien-Hsin S. Lee. Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3d die-stacked drams. In *Proceedings of the 40th Intl. Symp. on Microarchitecture*, 2007.

[26] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, 2009.

[27] Long Zhe Han, Yeonseung Ryu, Tae Sun Chung, Myungho Lee, and Sukwon Hong. An intelligent garbage collection algorithm for flash memory storages. In *Proceedings of the 6th international conference on Computational Science and Its Applications - Volume Part I*, ICCSA'06, 2006.

[28] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[29] I. Hur and C. Lin. Adaptive history-based memory schedulers. In *MICRO-37*, 2004.

[30] I. Hur and C. Lin. A comprehensive approach to dram power management. In *HPCA*, 2008.

[31] Intel Corporation. First the Tick, Now the Tock: Next-Generation Intel Microarchitecture (Nehalem). `http://www.intel.com/technology/architecture-silicon/next-gen/whitepaper.pdf`.

[32] E. Ipek, O. Mutlu, J. F. Martinez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA*, 2008.

[33] JEDEC. Ddr-4 sdram specification. `http://www.jedec.org/standards-documents/results/jesd79-4\%20ddr4`, 2012.

[34] Seunggu Ji and Dongkun Shin. An efficient garbage collection for flash memory-based virtual memory systems. In *IEEE Transactions on Consumer Electronics*, volume 56, November 2010.

[35] F Kashfi and S. Mehdi Fakhraie. Implementation of a high speed low-power 32 bit adder in 70nm technology. In *ISCAS*, 2006.

[36] Charles A. KILMER, Kyu-hyoun KIM, Warren E. MAULE, and Vipin PATEL. Memory system with a programmable refresh cycle, 06 2012.

[37] H.J. Kim and S.G. Lee. An effective flash memory manager for reliable flash memory space management. In *IEICE Transactions on Information Systems*, 2002.

[38] Youngjae Kim, Brendan Tauras, Aayush Gupta, Dragos Mihai, and Nistor Bhuvan Urgaonkar. Flashsim: A simulator for nand flash-based solid-state drives. In *First International Conference on Advances in System Simulation*, 2009.

[39] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

[40] Ohhoon Kwon, Kern Koh, Jaewoo Lee, and Hyokyung Bahn. FeGC: An efficient garbage collection scheme for flash memory based storage systems. In *Journal of System Software*, 2011.

[41] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power aware page allocation. In *ASPLOS-IX*, 2000.

[42] Sungjin Lee, Dongkun Shin, and Jihong Kim. BAGC: Buffer-Aware Garbage Collection for Flash-Based Storage Systems. In *11th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, 2012.

[43] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller. Energy management for commercial servers. *Computer*, 36(12):39–48, 2003.

[44] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. Raidr: Retention-aware intelligent dram refresh. In *ISCA*, 2012.

[45] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. Flikker: Saving dram refresh-power through critical data partitioning. In *ASPLOS*, 2011.

[46] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in smt processors. In *ISPASS*, 2001.

[47] M-Systems. Flash-memory Translation Layer for NAND flash (NFTL).

[48] M-Systems. TrueFFS: Wear Leveling Mechanism.

[49] Dongzhe Ma, Jianhua Feng, and Guoliang Li. Lazyftl: A Page-Level Flash Translation Layer Optimized for NAND Flash Memory. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, 2011.

[50] C. Manning and Wookey. YAFFS specification. In *Aleph One Limited*, 2001.

[51] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, 1996.

[52] Janani Mukundan and José F. Martínez. MORSE: Multi-objective reconfigurable self-optimizing memory scheduler. In *HPCA*, 2012.

[53] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *ISCA-35*, 2008.

[54] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO-40*, 2007.

[55] Eyec Hyun Nam, B.S.J. Kim, Hyeonsang Eom, and Sang-Lyul Min. Ozone (o3): An out-of-order flash memory controller architecture. In *IEEE Transactions on Computers*, 2011.

[56] Stefania Perri and Pasquale Corsonello. Fast low-cost implementation of single-clock-cycle binary comparator. In *ISCAS*, 2008.

[57] J. Pisharath, Y. Liu, W. Liao, A. Choudhary, G. Memik, and J. Parhi. NU-MineBench 2.0. Technical report, Northwestern University, August 2005. Tech. Rep. CUCIS-2005-08-01.

[58] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

[59] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA-27*, 2000.

[60] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 2011.

[61] SmartMedia Specification. SSFDC Forum, 1999.

[62] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *ASPLOS*, 2000.

[63] S. P. Song. Method and system for selective dram refresh to reduce power consumption. United States Patent #6094705, 2000.

[64] Jeffrey Stuecheli, Dimitris Kaseridis, Hillery C. Hunter, and Lizy K. John. Elastic refresh: Techniques to mitigate refresh penalties in high density memory. In *MICRO*, 2010.

[65] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis. Micro-pages: Increasing dram efficiency with locality-aware data placement. In *ASPLOS*, 2010.

[66] R. Sutton. Generalization in reinforcement learning. successful examples using sparse coarse coding. In *Neural Information Processing Systems Conference*, 1996.

[67] R. Sutton and A. Barto. *Reinforcement Learning*. MIT Press, Cambridge, MA, 1998.

[68] Ravi K. Venkatesan, Stephen Herr, and Eric Rotenberg. Retention-aware placement in dram (rapid): software methods for quasi-non-volatile dram. In *HPCA*, 2006.

[69] Yi Wang, Duo Liu, Meng Wang, Zhiwei Qin, Zili Shao, and Yong Guan. RNFTL: A Reuse-Aware nAND Flash Translation Layer for Flash Memory. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, LCTES '10, 2010.

[70] Yu-Shun Wang, Min-Han Hsieh, Chia-Ming Liu, Yi-Chi Wu, Bing-Feng Lin, Hsien-Chen Chiu, and Charlie ChungPing Chen. A 1.2v 6.4ghz 181ps 64-bit cd domino adder with dll measurement technique. In *ISCAS*, 2011.

[71] "Wear Leveling in Single Cell NAND Flash Memories". STMicroelectronics application note (AN1822), 2006.

[72] D. Woddhouse. JFFS: The journalling flash file system. In *Proocedings of Ottawa Linux Symposium*, 2001.

[73] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA-22*, 1995.

[74] Jon Worrel. Intel to introduce DDR4 memory with Haswell-EX server platform. In *fudzilla.com*, April 2012.

[75] Jung Yoon and Gary Tressler. Advanced flash technology status, scaling trends and implications to enterprise ssd technology enablement. In *Flash Memory Summit*, 2012.

[76] Z. Zhu and Z. Zhang. A performance comparison of dram memory system optimizations for smt processors. In *HPCA-11*, 2005.